Universidad de Alcalá

Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Diseño e implementación de un sistema de simulación electromagnética en remoto aplicado al análisis de antenas

Autor: José Luis García Díaz

Director: Álvaro Somolinos Yagüe

Tribunal:

Presidente: .....

Vocal 1º: .....

Vocal 2º: .....

Fecha: .....

Agradecimientos y dedicatoria

A mi familia, por haberme apoyado siempre.

Y a mis amigos de la carrera que han permanecido conmigo durante toda la carrera.

A todos... ¡Gracias!

Índice general

Índice de figuras

Índice de tablas

Resumen

Resumen

Palabras clave: Simulación electromagnética, software, cliente-servidor, Java

En esta memoria se presentará el trabajo realizado sobre la herramienta software *newFASANT* (versión 6). *newFASANT* es una herramienta informática para el análisis complejo de antenas, antenas a bordo de plataformas, RCS, estructuras periódicas, dispositivos de microondas, circuitos planos y electromagnéticos.

La herramienta consta de dos partes principales: interfaz gráfica y núcleos de simulación. La interfaz gráfica es una interfaz entre el usuario, quien proporciona los parámetros de la simulación, y los núcleos de simulación, que realizan los cálculos intensivos necesarios para generar los resultados que desea el usuario.

El trabajo realizado consiste en una adaptación de la herramienta a un paradigma cliente-servidor que permita la ejecución de simulaciones electromagnéticas en máquinas remotas, de forma que el usuario pueda utilizar la interfaz gráfica de forma local, pero realizar los cálculos intensivos en una máquina remota.

Dado que estos cálculos son muy costosos, la funcionalidad desarrollada en este proyecto podrá ser utilizada por usuarios con ordenadores de pocos recursos para delegar los cálculos a máquinas de mayores prestaciones. La solución aportada será completamente multiplataforma, sencilla, fácil e intuitiva de usar para los usuarios.

Abstract

Keywords: Electromagnetic simulation, software, client-server, Java

This document will show the work done on the *newFASANT* software tool (version 6). *newFASANT* is a computer tool for complex analysis of antennas, on-board antennas, RCS, periodic structures, microwave devices, planar and electromagnetic circuits.

The tool is composed of two main parts: graphical interface and simulation kernels. The graphical interface is an interface between the user, who provides the simulation parameters, and the simulation kernels, that perform the intensive calculations needed to generate the results the user wants.

The work described in this document consists on an adaptation of the tool to a client-server paradigm that allows execution of electromagnetic simulations in remote machines, so that the user can use the graphical user interface locally but perform the intensive calculations in the remote machine.

Given the complexity of the calculations, the developed functionality in this project can be used by users that own computers with little processing power in order to delegate the calculations to higher performance computers. This project seeks to adapt the existing software tool called *newFASANT* (major version 6), to a client-server paradigm that allows running remote EM simulations on remote machines. This, in turn, will allow clients to delegate the computationally costly process implied by a EM simulation to computers that have more computational resources than the client machines. The developed solution will be fully cross-platform, simple, easy and intuitive for the users.

Resumen extendido

El proyecto desarrollado consiste en el desarrollo de una nueva funcionalidad de la herramienta software *newFASANT*, producto software propietario y desarrollado por la empresa *NewFasant S.L.* Esta herramienta permite un amplio abanico de soluciones de simulación electromagnética, siendo destacables dos categorías: simulaciones relacionadas con RCS (*Radar Cross Section*) y con antenas (embarcadas en satélites, aviones y otras estructuras). En este trabajo nos centraremos en el desarrollo de la funcionalidad citada enfocada a la segunda categoría de simulaciones.

La funcionalidad desarrollada en este proyecto consiste en un módulo cliente-servidor que permita la ejecución de simulaciones relacionadas con antenas de forma remota, es decir, en una máquina distinta de la que es utilizada para la configuración de parámetros de la simulación (diseño de la geometría, elección del algoritmo usado para la simulación, definición de antenas y otros parámetros de la simulación). El objetivo es permitir a los usuarios de la aplicación utilizar equipos que carezcan de la potencia necesaria para realizar simulaciones, delegando los cálculos en una máquina con más recursos computacionales.

Este desarrollo aporta una solución más sencilla y fácil de usar que la ofrecida en la versión anterior de la herramienta *newFASANT* (versión 5), puesto que el proceso de comunicación entre cliente y servidor será manejado completamente por la aplicación usando el mecanismo de sockets de Java, de forma que además sea multiplataforma.

En este documento se habla, en primer lugar, del contexto en el que se desenvuelve el proyecto, comentando el origen de la herramienta *newFASANT* y describiendo los módulos de la herramienta con los que se trabaja en este proyecto de fin de grado. También se habla de los objetivos del proyecto y de sus antecedentes, contemplando las diferencias entre la solución desarrollada y la forma antigua de realizar las simulaciones en remoto. Tras esto, se habla del estado del arte, listando algunas de las herramientas de simulación electromagnética disponibles actualmente, así como algunas arquitecturas de trabajo remoto en otras aplicaciones similares. Por último, en este apartado se hablará de las herramientas utilizadas en el desarrollo del proyecto.

Para el desarrollo del sistema de simulación remota, se ha utilizado una metodología incremental, es decir, un desarrollo basado en iteraciones compuestas de un pequeño ciclo en cascada. En este documento se discutirán los distintos resultados de este proceso, incluyendo análisis de requisitos, diseño de la aplicación e implementación. También se describirá un caso práctico de uso de la funcionalidad desarrollada.

Por último, en este documento se detallan los recursos utilizados durante el desarrollo del proyecto y el presupuesto. Se incluye una guía de usuario que explica el modo de realizar simulaciones remotas, así como un manual de mantenimiento que explica cómo mantener el producto software, en vistas a un caso futuro en el que se desarrolle un nuevo módulo de la aplicación *newFASANT* y sea necesario ofrecer soporte remoto para el mismo.

Capítulo 1

Introducción

En este capítulo se hablará del entorno en el cual se desenvuelve el proyecto realizado, entrando en detalle sobre la herramienta *newFASANT*, así como de las motivaciones, objetivos y antecedentes del proyecto.

1.1 Contexto

En las últimas décadas, nuestra vida se ha visto facilitada gracias a los avances en el campo del electromagnetismo y las antenas. Prueba de ello son inventos como la televisión por satélite, la telefonía móvil o la radio, que son posibles gracias al uso de antenas para la recepción y transmisión de ondas electromagnéticas.

No obstante, el diseño de sistemas complejos que hagan uso de antenas puede ser bastante complicado, sobre todo si tenemos en cuenta la gran cantidad de parámetros que definen una antena: ancho de banda, directividad, ganancia, eficiencia, impedancia, etc. Adicionalmente, el entorno en el que se encuentra una antena también influye en su efectividad: si se encuentra en el espacio, aislada, tendrá un comportamiento distinto al que tendría si se encontrase rodeada de una estructura, como por ejemplo, un radomo. Por tanto, hay una gran cantidad de factores a tener en cuenta a la hora de diseñar un sistema complejo de antenas, lo que hace difícil medir la efectividad y corrección de dichos sistemas.

Por este motivo, para facilitar el diseño y prueba de sistemas electromagnéticos, al igual que en otras disciplinas, se utilizan herramientas de diseño asistido y de simulación que permiten modelar el sistema electromágnetico, introduciendo los parámetros que lo definen, y simular su funcionamiento. De esta forma, se ahorra mucho tiempo debido a la facilidad de cambiar los parámetros del sistema usando la herramienta software, y se ahorra dinero debido a que dichas pruebas no se realizan usando medios materiales (ahorrando costes de fabricación o adquisición de componentes y en cámaras anecoicas).

Desde el año 1995, el Grupo de Electromagnetismo Computacional (GEC) de la Universidad de Alcalá de Henares ha estado trabajando en el desarrollo de herramientas software dedicadas a la simulación electromagnética. Inicialmente, las herramientas de simulación electromagnéticas desarrolladas eran poco intuitivas, y era necesario que el usuario dispusiera de conocimientos avanzados sobre informática y sobre la herramienta, pues muchas de las tareas que era necesario realizar había que hacerlas de forma manual (como preparar los ficheros con los parámetros de la simulación), o usando herramientas externas (por ejemplo, para el diseño de la geometría). Poco a poco, estas herramientas han ido progresando en términos de experiencia de usuario y de funcionalidad, ofreciendo interfaces gráficas de usuario más intuitivas y atrayentes, y automatizando muchas de las tareas que antes debía hacer el operario a mano. De esta forma se consigue reducir el tiempo necesario para configurar una simulación y el requerimiento de conocimientos de la herramienta que debía tener el usuario [1].

En noviembre de 2010, a iniciativa del grupo de investigación antes mencionado, se fundó la empresa de base tecnológica (EBT) *NewFasant S.L* [2]. Esta empresa se encarga de desarrollar y comercializar la herramienta software *newFASANT*, la cual es una herramienta de simulación para análisis de campos electromagnéticos que ofrece una gran variedad de posibilidades de simulación. En efecto, esta herramienta se compone de varios productos (también llamados "módulos"), cada uno encargado de diferentes tareas, y clasificados en función de si se utilizan para calcular RCS o antenas.

Los módulos encargados de simular antenas, implementados en la versión 6 de la herramienta en el momento de la redacción de este documento, son los siguientes [4]:

- **GTD**, acrónimo procedente de *Geometrical Theory of Diffraction* o, en español, Teoría Geométrica de la Difracción. Este módulo se basa en la aplicación de la teoría que le da nombre, así como de la Óptica Geométrica y la Óptica Física, para el análisis de antenas embarcadas en satélites, aviones y otros modelos complejos.

- **MoM**, acrónimo procedente de *Method of Moments* o, en español, Método de los Momentos. Este módulo, además de utilizar el método de los momentos que le da nombre, hace uso de las técnicas FMLMP (*Fast Multilevel Multipole*), CBFs (*Macro Basis Functions*) y decomposición de dominios para el análisis eficiente en tres dimensiones de antenas complejas, antenas embarcadas, sección radar y compatibilidad electromagnética.

El proyecto realizado, y el cual se describirá en este documento, trata del análisis, diseño e implementación de la funcionalidad de simulación remota centrándose en los módulos anteriores. No obstante, gran cantidad del trabajo realizado se puede aplicar al resto de módulos de la herramienta *newFASANT*.

1.2 Motivación

La estructura de la herramienta software *newFASANT* muestra una división en dos capas. Por un lado, se tiene un frontend desarrollado en el lenguaje de programación Java que consta de una interfaz gráfica de usuario (GUI) y, por otro lado, un backend con los núcleos de simulación y el mallador. Los núcleos de simulación y el mallador están desarrollados en FORTRAN, y son los encargados de realizar los cálculos intensivos necesarios para llevar a cabo la simulación y generar los resultados.

El rol de la primera de las capas, el frontend, es el de hacer de intermediario entre el usuario y los núcleos de simulación, ofreciendo al usuario una interfaz amigable con la cual puede configurar los parámetros de la simulación, los cuales dependerán del producto o módulo que se esté usando. Según estos parámetros y el módulo utilizado, el frontend genera unos archivos que reflejan los distintos parámetros de la simulación, y que son requeridos para el funcionamiento de los núcleos de simulación. El frontend, a continuación, llama al núcleo encargado de la simulación (existen tres núcleos, además del mallador, que son el kernel *MONURBS*, el kernel *POGCROS* y el kernel *FASANT*). El núcleo realiza los cálculos pertinentes, tomando como datos de entrada el contenido de los ficheros creados por el frontend Java. Por último, el kernel escribe los ficheros que contienen los resultados de la simulación, y que son leídos por el frontend para presentar dichos resultados al usuario de una forma vistosa e intuitiva.

Los núcleos de simulación y el mallador son programas, independientes de la interfaz desarrollada en Java, que son bastante eficientes en sus cálculos debido al uso de tecnologías de procesamiento paralelo como son MPI (interfaz de paso de mensajes utilizada para la comunicación de procesos en ejecución en distintos procesadores), OpenMP (API de programación paralela en sistemas de memoria compartida) y CUDA (framework utilizado para explotar la potencia de cálculo en paralelo en los núcleos de las tarjetas gráficas NVIDIA), así como debido al uso de algoritmos eficientes para realizar dichos cálculos.

No obstante, en muchas ocasiones, realizar los cálculos de una simulación puede ser muy costoso en términos computacionales, sobre todo si se trata con escenarios complejos, con mucho detalle o con muchos elementos. En estos casos, la llamada al kernel o al mallador puede resultar en un cuello de botella importante en el proceso de realización de la simulación, tardando mucho más tiempo del que al usuario le gustaría, sobre todo en máquinas menos potentes.

Esto motiva la construcción de un sistema que permita diseñar el entorno y configurar los parámetros del escenario a simular en ordenadores con pocos recursos, pero puedan delegar el trabajo intensivo que realizan los núcleos de simulación y el mallador en máquinas que dispongan de mayor cantidad de recursos computacionales, como mayor número de procesadores, procesadores más potentes, mayor cantidad de memoria principal con la que poder trabajar y/o tarjetas gráficas que permitan un procesamiento paralelo más avanzado.

1.3 Objetivos

El objetivo principal es diseñar e implementar un sistema distribuido, enfocado a los módulos GTD y MoM de la herramienta *newFASANT*, que permita a los usuarios (clientes) configurar el entorno y los parámetros relativos a una simulación y ejecutar dicha simulación en una máquina remota (servidor). El sistema también deberá hacer que la máquina remota devuelva al cliente los resultados resultantes de ejecutar la simulación, para que éste pueda mostrar dichos resultados en su interfaz gráfica.

Además de este objetivo principal, se proponen los siguientes objetivos secundarios, complementarios al principal:

- Analizar el funcionamiento e implementación de la aplicación *newFASANT* en su versión 6.
- Desarrollar un sistema que sea completamente multiplataforma, es decir, que funcione correctamente en entornos Windows y UNIX.
- Diseñar e implementar un protocolo que haga posible la comunicación entre cliente y servidor a través de la red.
- Ofrecer una interfaz gráfica de usuario intuitiva para la configuración de los parámetros de las máquinas remotas (dirección de red y puerto) en el cliente.
- Maximizar la reutilización de código entre las implementaciones para los distintos módulos, de forma que no se repita el trabajo a la hora de implementar la funcionalidad remota para módulos diferentes.
- Diseñar el sistema teniendo en cuenta, en la medida de lo posible, los principios y técnicas de diseño de la programación orientada a objetos. Además, se considerará el uso de patrones de diseño para lograr un diseño más fácil de mantener y entender.
- Para facilitar la integración de la funcionalidad en el programa, así como reducir al mínimo la posibilidad de introducir defectos nuevos en el programa, se intentará modificar la menor cantidad posible del código ya existente.

Como objetivo adicional, se incluye la redacción de esta memoria, la cual plasmará por escrito el proceso de desarrollo de la funcionalidad y la explicación sobre su diseño e implementación, y ofrecerá una guía de mantenimiento para adaptar los módulos que se desarrollen en un futuro a la funcionalidad desarrollada, así como un manual de usuario.

1.4 Antecedentes

Anteriormente al desarrollo de este proyecto de fin de grado, no existía ninguna forma de realizar simulaciones en máquinas remotas usando la versión 6 de la herramienta *newFASANT*. Previamente, si se deseaba realizar la simulación en otra máquina distinta a la cual se configuraba el entorno de la simulación, era necesario transferir los ficheros del proyecto a la máquina remota usando una herramienta de transferencia de archivos, como

el `scp` de sistemas UNIX. Posteriormente, se podría invocar a los núcleos de simulación o mallador de forma remota usando un cliente de shell remota, como `ssh` (en UNIX) o PuTTY (en Windows). Tras acabar la operación de simulación, se copian los resultados a la máquina local usando `scp` o PuTTY.

El proyecto realizado intenta evitar este proceso tan complejo, integrando la lógica de simulaciones remotas dentro de la propia aplicación y adaptándola que funcione de una manera transparente para el usuario.

Capítulo 2

Estado del arte

En este capítulo, se entrará más en detalle sobre el funcionamiento de la herramienta *newFASANT*, analizando un caso de uso de la misma. Se explicará como se realiza la interacción entre la interfaz de la aplicación, desarrollada en Java, y los núcleos de simulación. Posteriormente, se discutirán algunos modelos de trabajo en máquinas remotas usados por software profesional. Por último, se hablará de las tecnologías utilizadas en el desarrollo del proyecto a tratar.

2.1 Software de simulación electromagnética

Como se describe en el capítulo anterior, la gran cantidad de factores que existen en sistemas complejos de telecomunicaciones hace necesario el uso de herramientas software que asistan a los diseñadores de dichos sistemas en su análisis. Así se evitan pérdidas de dinero y tiempo en pruebas realizadas con prototipos o, en el peor de los casos, en arreglar sistemas que hayan sido puestos en producción.

Actualmente, se encuentran disponibles varias de estas herramientas de simulación, tanto libres como propietarias, que permiten simular antenas. Entre estas herramientas se incluyen:

- *Feko*: Propietaria, aunque tiene aplicación en múltiples campos del electromagnetismo (no solo en antenas) y ofrece un gran número de posibilidades [5].
- *HFSS*: Herramienta software propietaria de simulación en 3D de campos electromagnéticos de onda completa y diseño de alta frecuencia y de componentes de alta velocidad [6].
- *CST Studio Suite*: Conjunto de herramientas capaces de realizar simulaciones electrostáticas y magnetostáticas, estacionarias y de baja y alta frecuencia, así como de calcular los efectos de campos electromagnéticos sobre diversas sustancias y materiales [7].

- *MMANA-GAL*: Gratuita. Herramienta de análisis de antenas basada en el método de los momentos, aunque con ciertas limitaciones [8].
- *4nec2*: Modelador de antenas en 2D y 3D gratuito que permite analizar patrones de radiación de campo cercano y campo lejano [9].

La herramienta *newFASANT* se compone de dos módulos para el trabajo con antenas: MoM y GTD, que permiten realizar simulaciones de distintos tipos. Estos productos usan, como base, sus núcleos de simulación para realizar los cálculos y utilizan algoritmos avanzados para mejorar la eficiencia de los mismos. En el siguiente apartado se describe un caso de uso mostrando el proceso de simulación que realizaría un usuario de la herramienta.

2.2 La herramienta newFASANT

La herramienta *newFASANT* se compone de una interfaz de usuario, hecha en Java, que además de ofrecer una interfaz amigable para que el usuario pueda configurar la simulación, se encarga de realizar los pasos necesarios para la correcta ejecución de los núcleos de simulación. El proceso típico para realizar una simulación es el siguiente:

1. Se ejecuta la herramienta y se crea el proyecto. El usuario debe especificar el módulo utilizado en función del tipo de simulación que desee realizar. En este documento, únicamente se tendrán en cuenta aquellos que traten con antenas.
2. El usuario crea el escenario que se utilizará en la simulación. Para ello, se debe modelar la geometría usando superficies paramétricas llamadas NURBS.
3. El usuario define los parámetros de simulación. Estos parámetros son los siguientes:
   - Parámetros de simulación, como la frecuencia o rango de frecuencias de la simulación, tipo de simulación y otras opciones especificas del módulo (como elección de un algoritmo específico).
   - Antenas, cuyos parámetros incluyen: tipo de antena (dipolo, patrón de radiación o multipolo), posición dentro de la geometría, amplitud, fase y otros parámetros dependientes del tipo de la antena (por ejemplo, número de dipolos eléctricos y magnéticos en el caso de antenas de dipolo).
   - Parámetros de observación, que determinarán los puntos del espacio en los que se calcularán los valores del campo.
4. El usuario ordena el mallado de la geometría al mallador. El mallado es un proceso que toma como entrada la geometría, definida de forma paramétrica, y la procesa dividiendo las superficies en formas sencillas, como triángulos o cuadriláteros, de forma que se deja en un formato (malla) adecuado para que sea posible aplicarle el algoritmo de simulación electromagnética [3].
5. El usuario ordena la ejecución de la simulación por parte del núcleo de simulación correspondiente. Para conseguir esto, la interfaz escribe una serie de ficheros con

los parámetros necesarios para la simulación y el núcleo de simulación lee estos ficheros para obtener dichos parámetros, definidos por el usuario. El núcleo de simulación, tras terminar la ejecución de la simulación, habrá escrito una serie de ficheros con los resultados.

6. El usuario visualiza los resultados de la simulación. Para ello, la interfaz lee los ficheros escritos por el núcleo de simulación y muestra su contenido de forma visual e intuitiva, sea mediante gráficas o mediante diagramas en 3D.

En la Figura 2.1 se puede observar, de manera gráfica, el proceso descrito, donde los recuadros redondeados corresponden a acciones iniciadas por el usuario.

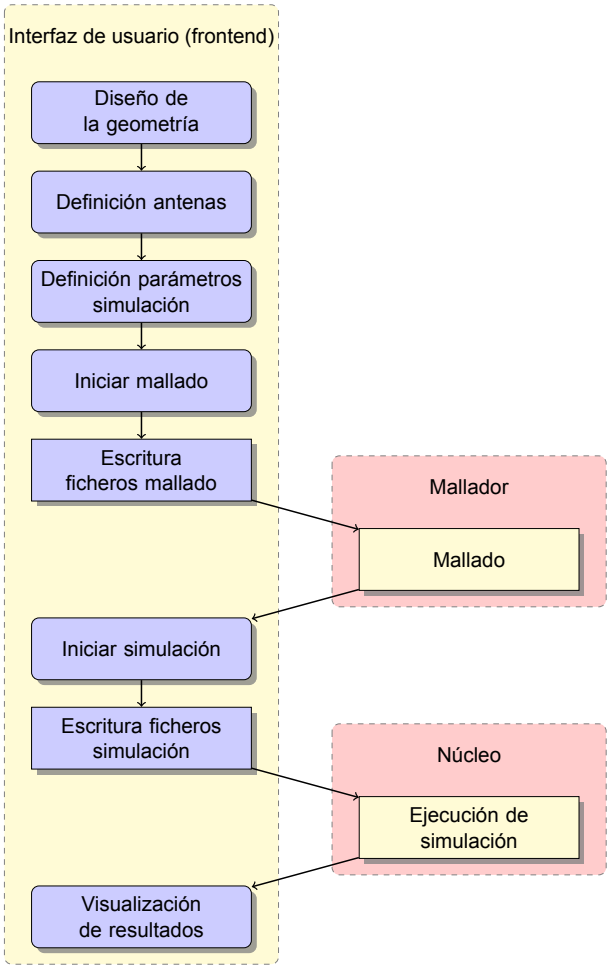

Figura 2.1: Diagrama representando el flujo de una simulación en newFASANT

La separación entre el mallado y la ejecución de la simulación es importante, pues si el usuario desea modificar algún parámetro de la simulación sin modificar la geometría, no necesitará volver a mallar si ya lo ha hecho anteriormente: bastará con que vuelva a

ejecutar la simulación. De esta forma, es posible modificar los parámetros de la simulación tantas veces como se quiera sin la necesidad de invocar al mallador cada vez, permitiendo ahorrar tiempo en cálculos.

2.3 Arquitecturas de trabajo en remoto

Es bastante común, no solo en software de simulación electromagnética, sino en software de diseño asistido por computador en general, incluir la posibilidad de realizar las operaciones de configuración en una máquina cliente, típicamente de pocas prestaciones, y dejar aquellas operaciones que requieran la realización de cálculos intensivos a máquinas de más altas prestaciones. Este paradigma requiere que ambas máquinas se encuentren comunicadas por una red, usada para el intercambio de información. La Figura 2.2 representa el más sencillo de estos sistemas distribuidos, donde existe una máquina cliente y una máquina servidora.

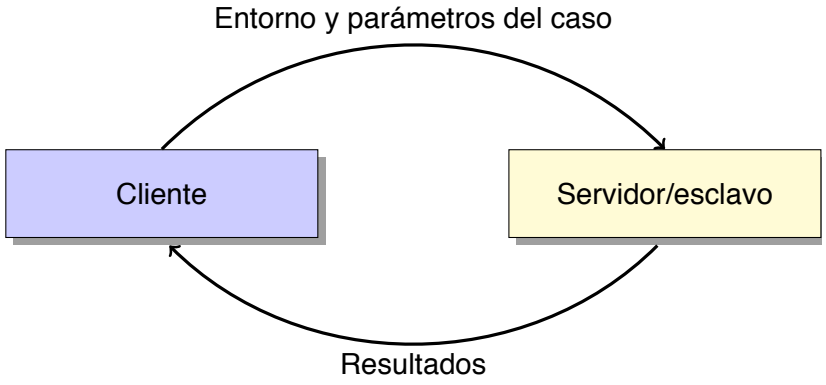

Figura 2.2: Diagrama representando el paradigma cliente-servidor

Un ejemplo de este tipo de herramientas es Blender: software de diseño y animación de escenas en tres dimensiones y de código abierto. Blender permite realizar el renderizado[1] de una animación en una máquina o conjunto de máquinas remotas. Este proceso es conocido popularmente como *network rendering*, y se basa en una arquitectura en red centralizada con tres roles diferentes:

- **Cliente**: Máquina que dispone de los datos del escenario a renderizar. Se comunica con el maestro, y nunca con los esclavos.
- **Maestro**: Máquina que hace el rol de servidor. Se encarga de recibir los trabajos de los clientes y de distribuirlos entre los distintos esclavos. Actúa de intermediario, enviando los datos de los escenarios a renderizar desde los clientes a los esclavos.

[1] El renderizado es el proceso de generar una imagen o vídeo por computador, mediante la realización de cálculos intensivos (sobre todo de iluminación) a partir de un modelo geométrico en 3D.

Cuando se termina de renderizar la animación, recibe los resultados de los esclavos (las imágenes renderizadas) y se las envía al cliente que solicitó el renderizado.

- **Esclavo**: Rol de las máquinas que disponen de los recursos de computación. Se comunican con el maestro, recibiendo de este los datos de los escenarios a renderizar, y enviando de vuelta las imágenes renderizadas.

El objetivo de esta arquitectura en tres niveles es el de permitir la cooperación entre múltiples clientes y múltiples esclavos, con la figura del "maestro" intercediendo entre ambos grupos y coordinando el uso de los esclavos por parte de los clientes mediante la distribución de trabajos. Esta arquitectura, mostrada en la Figura 2.3 permite, por tanto, un trabajo completamente multiusuario.

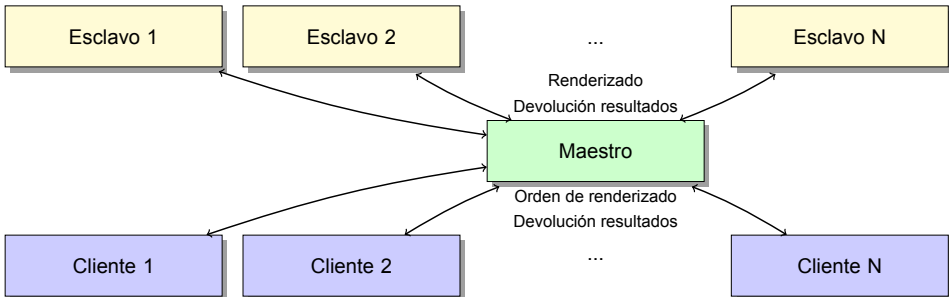

Figura 2.3: Diagrama sobre la arquitectura de *Network rendering* de Blender

Otras soluciones profesionales (por ejemplo, *Feko*) ofrecen servicios de computación en la nube con el objetivo de ofrecer acceso a sus clientes a equipos de altas prestaciones con su software instalado. De esta forma, los usuarios pueden ejecutar sus simulaciones en una máquina remota ubicada en algún lugar de Internet disponiendo únicamente de la interfaz del programa, y sin necesidad de disponer de hardware avanzado (y por tanto, caro) en el lado del cliente [5].

La solución que se ha desarrollado en este trabajo de fin de grado se asemeja a la mostrada en la Figura 2.2. El objetivo es que la misma aplicación pueda operar tanto en modo cliente como en modo servidor, permitiendo una opción u otra a la hora de iniciar la aplicación mediante el uso de parámetros de línea de comandos. La solución ofrecida no será multiusuario, debido a la limitación del programa en su versión actual que impide realizar más de una simulación en el mismo. No obstante, se realizará una mejora sobre el sistema planteado para solventar parcialmente este inconveniente que permitirá una arquitectura distribuida similar a la de Blender, anteriormente mostrada, aunque sin el rol del maestro.

2.4 Tecnologías utilizadas

En esta sección se habla de las tecnologías y las herramientas software utilizadas para el desarrollo del proyecto, así como de los motivos que han llevado a la elección de

estas tecnologías.

2.4.1 Lenguaje de programación Java

Java es una plataforma de ejecución multiplataforma desarrollada inicialmente por *Sun Microsystems* y que es actualmente propiedad de *Oracle Corporation*. La plataforma Java se compone de una máquina virtual (*Java Virtual Machine* o *JVM*), un lenguaje de programación (lo que entendemos como el "lenguaje de programación Java") y diversas herramientas para el desarrollador, siendo las más destacables el compilador y el intérprete. Todos los elementos anteriores se ponen en disposición del desarrollador en el JDK (el kit de desarrollo de Java).

El lenguaje Java se conoce como un lenguaje semi-interpretado debido a que el proceso de ejecución de un programa Java implica una fase de compilación, la cual genera un código intermedio (*bytecode*) a partir del programa Java (que no llega a ser código máquina) y una fase de interpretación que toma el código intermedio anterior y lo ejecuta en la máquina virtual de Java. Dado que el programa Java no es completamente compilado ni completamente interpretado, se cataloga al lenguaje de programación como una mezcla de ambos tipos y por tanto se le da la denominación de "semi-interpretado". Este esquema de ejecución facilita la portabilidad de la que presume Java, ya que de esta forma, cualquier programa Java compilado podrá ejecutarse en cualquier ordenador que disponga de una máquina virtual de Java, siempre que dicha máquina virtual cumpla la especificación.

Debido a que este proyecto trata de la adaptación de un software realizado en Java, la elección de este lenguaje de programación para la implementación del mismo es inevitable. No obstante, Java es un lenguaje adecuado para este proyecto debido a la facilidad de uso de las APIs de sockets y de trabajo con lenguajes XML, que se explicarán a continuación. Además, el lenguaje incorpora un conjunto de funcionalidades que permiten, de forma sencilla, la gestión de la concurrencia y una fácil creación y gestión de tareas e hilos. Por último, aunque no menos importante, se dispone de experiencia en el manejo de este lenguaje de programación, debido a que ha sido utilizado en un alto número de asignaturas del plan de estudios.

2.4.2 Paradigma cliente-servidor, sockets y protocolo TCP

En el diseño de una aplicación distribuida, es necesario pensar en el modelo de red utilizado, es decir, qué máquinas formarán parte de la red y qué comunicaciones hay entre ellas. En este aspecto, existe un modelo de red ampliamente utilizado: el **modelo cliente-servidor**.

Este modelo de aplicación distribuida se basa en la existencia de dos roles: cliente y servidor. El cliente es toda aplicación que solicite un servicio, y un servidor es una aplicación que dispone de los recursos necesarios de ofrecer dicho servicio. De esta forma,

se puede conseguir una cooperación entre el cliente y el servidor: el cliente solicita a la aplicación servidor un determinado servicio, a lo que servidor responde al cliente con los recursos solicitados, resultantes de la ejecución de dicho servicio. La aplicación cliente puede encontrarse en la misma máquina que la aplicación servidor, aunque normalmente no suele ser así.

En el proyecto desarrollado, este modelo de aplicación distribuida encaja perfectamente ya que ambos roles están perfectamente identificados: por un lado, el cliente consiste en la interfaz gráfica utilizada por el usuario para configurar la simulación y, por otro lado, el servidor, que es la aplicación ejecutándose en la máquina donde se ejecutará la simulación, es la encargada de ofrecer el servicio de realizar dicha simulación y devolver los resultados (recursos resultantes de la ejecución del servicio) al cliente.

Para poder establecer un canal de comunicación entre la máquina cliente, en la cual se encuentran los datos de la simulación a llevar a cabo, y la máquina servidor, la cual dispone de los recursos de computación deseados para ejecutarla, es necesario hacer uso de algún protocolo de transporte. Los dos protocolos de transporte más significativos son:

- **Protocolo UDP** (*User Datagram Protocol*). Se le conoce por ser un protocolo de transporte "no fiable" en tanto que no ofrece ninguna garantía sobre la integridad de los paquetes, ni sobre la llegada de los paquetes en el mismo orden que fueron enviados. De hecho, no garantiza siquiera su llegada. Tiene la ventaja de que, al no incluir las garantías anteriores, es más eficiente.
- **Protocolo TCP** (*Transmission Control Protocol*). Se trata de un protocolo de transporte "fiable" dado que, a diferencia de UDP, sí implementa mecanismos para garantizar la integridad y llegada en orden de los mensajes. Al implementar estas garantías, el protocolo hace uso de comprobaciones adicionales y, por tanto, es menos eficiente que UDP, pero permite al programador abstraerse de los problemas que ocurren en la red y centrarse en la comunicación.

El protocolo de transporte utilizado en la implementación de la comunicación en este proyecto es TCP por la razón ya comentada: fiabilidad en la transmisión de datos. En este caso, UDP no es una opción viable ya que la información a transferir (los datos de la simulación y los resultados) es de gran tamaño[2] y por tanto, cualquier alteración en esta información debido a incidencias en la red puede ocasionar problemas a la hora de cargar el proyecto en la máquina destino. Para solventar este problema, es necesario disponer de las comprobaciones de integridad (sumas de comprobación) y de secuencialidad (números de secuencia que aseguran la llegada en orden de los paquetes a la aplicación) que implementa TCP.

No obstante, el protocolo de transporte TCP solo se encarga de la transmisión y recepción de datos entre dos procesos (ubicados o no en máquinas diferentes). Esto quiere decir que es necesario implementar un protocolo de aplicación, que delegue en

[2]Según la complejidad del proyecto a simular, el tamaño del mismo puede llegar al orden de los megabytes.

TCP la transmisión de datos y que sí tenga en cuenta la semántica de los datos que se estén transfiriendo. De esta forma, tanto la aplicación cliente como la aplicación servidor pueden comunicarse y "comprender" lo que la otra está diciendo. Se hablará del diseño de dicho protocolo en el Capítulo 4.

Por último, dentro del plano de la programación, es necesario proporcionar al desarrollador de un API (interfaz de programación de aplicaciones) para el uso de estos protocolos de transporte, esto es, realizar el intercambio de datos entre las máquinas que se comunican. La gran mayoría de lenguajes de programación permiten el uso de una abstracción que hace esto posible: **los sockets**. Un socket es un objeto que debe ser configurado con los parámetros de la conexión (como mínimo: protocolo a usar, dirección de red del proceso remoto y número de puerto remoto) y que, a partir de entonces, se puede utilizar para establecer una comunicación con dicho proceso. Un socket se trata de una *abstración* para el programador dado que, para él, es una caja negra: los detalles de la implementación de la comunicación son manejados de forma transparente por el entorno de ejecución y por el sistema operativo, permitiendo que el programador se despreocupe de la implementación del protocolo de transporte utilizado.

2.4.3 Lenguajes XML y biblioteca JAXB

En aplicaciones que hagan uso de información que debe ser persistente entre distintas ejecuciones, es necesario disponer de alguna forma de almacenar dicha información en un dispositivo de almacenamiento persistente. Este proceso es conocido por el nombre de **serialización** (o *marshalling*). Asimismo, es necesario ser capaz de recuperar dicha información cuando sea necesario disponer de ella, proceso que se conoce como **deserialización** (o *unmarshalling*).

Para hacer persistente la información, es necesario convertir las estructuras de datos que la contienen en un formato que permita su almacenamiento y posterior lectura en, o desde, un fichero. Existen, principalmente, dos tipos de formatos:

- *Binarios*: La información se guarda codificada en binario, siendo por lo general formatos poco legibles por el ser humano. No obstante, estos formatos consiguen una mayor eficiencia a la hora de escribir y leer la información, así como un menor tamaño de los ficheros.
- *De texto*: La información se guarda en ficheros de texto plano. Estos formatos tienen la ventaja de ser fácilmente entendibles por el ser humano, pero normalmente son más pesados (ya que los ficheros basados en texto codifican toda la información en forma de caracteres, lo cual ocupa más espacio que codificar dicha información en binario). Su decodificación puede ser ligeramente más lenta, puesto que es necesario convertir una representación textual al formato nativo de las estructuras de datos originales, lo cual puede implicar realizar un análisis sintáctico y semántico (*parsing*) para comprobar la validez del contenido del fichero a cargar y extraer su información.

En la implementación del proyecto, es necesario almacenar la información de los servidores de forma que no sea necesario introducir los datos del servidor donde se desea ejecutar la simulación en cada proyecto. Serializar esta información permitiría hacer persistente la misma y así poder mantenerla entre distintas ejecuciones del programa.

Para la serialización de la información, se ha optado por utilizar un formato basado en texto. Concretamente, se hará uso del **lenguaje XML** (*eXtensible Markup Language*). XML es un metalenguaje que se utiliza para definir lenguajes de marcado (basados en el uso de etiquetas). Esto quiere decir que XML permite al desarrollador construir su propio lenguaje de marcado, definiendo la jerarquía y tipos de datos que se almacenarán. Dependerá del programa el interpretar la información contenida en los ficheros XML y traducirla a estructuras de datos nativas.

Se ha decidido utilizar XML para la serialización de los datos ya que, al ser de formato textual, los ficheros pueden ser interpretados por seres humanos, facilitando la inspección de la información contenida en estos. A pesar de ser un formato textual, el aumento en tamaño de los ficheros y la sobrecarga introducida por el análisis sintáctico al analizar los ficheros son despreciables, ya que típicamente se trabajará con ficheros pequeños que no contienen gran cantidad de servidores.

El otro motivo por el cual se ha decidido utilizar XML, y no cualquier otra forma de almacenamiento en texto plano, es la existencia de la **biblioteca JAXB**. Las siglas JAXB provienen de *Java Architecture for XML Binding*, y es una biblioteca que forma parte de la plataforma de Java que permite convertir objetos de la aplicación Java a XML y viceversa. De esta forma, podemos delegar la serialización y la deserialización del fichero de configuración en esta biblioteca, permitiendo olvidarnos acerca de la implementación de dicha serialización y dejar el código más limpio. El modo en el que se ha empleado esta biblioteca se describirá en el Capítulo 5.

2.4.4 Diagramas UML

Con el fin de poder describir los resultados de cada una de las fases del desarrollo del software de manera gráfica, se utilizarán algunos tipos de diagramas UML. UML proviene de *Unified Modelling Language* y, como tal, es un lenguaje de modelado de sistemas software. A continuación se explicarán distintos tipos de diagramas que define el estándar UML que han sido utilizados en el desarrollo de este proyecto:

- **Diagramas de casos de uso**: Este tipo de diagramas, utilizado en la fase de análisis de requisitos, describe las interacciones entre los usuarios del sistema (conocidos como actores) y la aplicación. Estos diagramas representan las funcionalidades del programa mediante los *casos de uso* (dibujados en forma de óvalos), así como la relación entre las mismas y los actores involucrados.
- **Diagramas de secuencia**: Este tipo de diagramas pertenece a la fase de diseño y se ocupan de describir las interacciones entre los distintos elementos del programa a través del tiempo.

- **Diagramas de clases**: Este tipo de diagramas también pertenece a la fase de diseño y muestra la estructura estática de la aplicación; esto es, las clases de la aplicación y la relación entre ellas. Se dice que representa la estructura estática porque no muestra la comunicación de los objetos presentes durante la ejecución del programa, como sí hacen los diagramas de secuencia.

Capítulo 3

Especificación y análisis de requisitos

En este capitulo se listarán los requisitos de la aplicación y se hará un análisis de los mismos, realizando diagramas de casos de uso para mostrar gráficamente la relación entre los distintos casos de uso del proyecto.

3.1 Descripción detallada del sistema

En este proyecto, se desea adaptar la aplicación actual de *newFASANT* (versión 6) para que sea capaz de funcionar mediante un paradigma cliente-servidor. Es decir, la misma aplicación debe ser capaz de operar en modo cliente (la cual deberá mostrar interfaz gráfica) y en modo servidor (la cual se ejecutará en modo texto y atenderá las peticiones del cliente). No obstante, se tiene la restricción de que tiene que ser la misma aplicación la que funcione en cualquiera de los dos modos (cliente o servidor) según indique el usuario a la hora de lanzarla, en contraposición a la idea de crear un programa distinto que actúe únicamente como servidor. La justificación a esta restricción es reducir el coste de mantenimiento de dos aplicaciones separadas, así como promover la reutilización del código ya desarrollado de la aplicación original en la realización del módulo servidor.

A continuación se detallarán los roles de los dos componentes de la arquitectura en red cliente-servidor ideada:

3.1.1 Descripción del cliente

En este modo, la aplicación deberá funcionar, a grandes rasgos, de la misma forma en la que funciona actualmente. Esto quiere decir que puede seguir usándose la misma aplicación para ejecutar simulaciones en local. No obstante, deberá agregarse una opción

que permita al usuario seleccionar si desea realizar sus simulaciones en local o en remoto y, en este último caso, pueda realizar la configuración de los servidores a utilizar.

La aplicación debe permitir al usuario añadir la información de todos aquellos servidores a los que desee conectarse para realizar simulaciones. De cada servidor se deberá tener su dirección IP y el puerto donde la aplicación servidor está escuchando solicitudes, así como, opcionalmente, un nombre para identificar a ese servidor. Asimismo, el usuario debe ser capaz de eliminar un servidor dado de alta previamente si así lo desea.

Por otro lado, para que el usuario pueda comprobar que los datos introducidos son correctos, así como la disponibilidad del servidor introducido, se incorporará una opción que hará una comprobación a dicho servidor. Según esta comprobación, el usuario podrá saber si el servidor cuyos datos ha introducido está disponible o si, por el contrario, dicho servidor no está ejecutándose o se han introducido datos incorrectos.

Además, se añadirá una funcionalidad para la importación/exportación de una lista de servidores, de forma que sea más sencillo el intercambio de información de servidores entre distintos usuarios.

Una vez configurada la lista de servidores, el usuario podrá seleccionar uno de ellos para ejecutar en él las simulaciones. De esta forma, si hay un servidor seleccionado, se utilizará para llevar a cabo los procesos de mallado de la geometría y de cálculo de resultados. La ejecución de estos procesos en remoto será transparente para el usuario, en el sentido de que el usuario deberá realizar los mismos pasos que haría para realizar la simulación en local (a parte de la configuración y selección del servidor) y que el usuario podrá ver los resultados de la simulación de la misma forma que si dichos resultados se hubieran calculado en su propia máquina. Con esto, se pretende que realizar la simulación en remoto sea lo más intuitivo posible y facilitar a los usuarios el aprendizaje de la nueva funcionalidad.

Adicionalmente, para subsanar parcialmente el hecho de que un servidor no pueda ejecutar más de una simulación en paralelo y, por tanto, no ser multiusuario, se añadirá una opción adicional. Esta opción permitirá que si la aplicación cliente detecta que el servidor seleccionado no se encuentra disponible, ya sea porque esté ocupado realizando otra simulación o no se esté ejecutando un servidor en ese momento, la aplicación cliente intente conectarse al siguiente servidor de la lista y realizar la simulación en este servidor. Si dicho servidor tampoco se encuentra disponible, se prueba con el siguiente y el proceso continúa hasta que:

1. Se encuentra un servidor que es capaz de responder a la solicitud. En este caso, se envían los datos de la simulación a ese servidor y se realizan los cálculos pertinentes de la simulación en el mismo.

2. No se encuentra ningún servidor capaz de responder a la solicitud. En este caso, la simulación es abortada y se muestra un mensaje de error al usuario indicando la incidencia.

A esta característica de probar diferentes servidores hasta que se encuentre uno disponible (o no se encuentre ninguno) la llamaremos "*server fallback*"[1].

Por último, mencionar que se trata de permitir esta funcionalidad en los módulos GTD y MoM de la herramienta, aunque se desea que la mayor parte del trabajo realizado pueda aplicarse al resto de los módulos.

3.1.2 Descripción del servidor

En el modo servidor, la aplicación no deberá mostrar interfaz gráfica de usuario puesto que este modo no está planteado para su uso directo por los usuarios. En su lugar, deberá actuar como una aplicación de línea de comandos que deberá indicar en su salida cada uno de los eventos que suceden en la aplicación servidor.

Cuando la aplicación se esté ejecutando en modo servidor, deberá escuchar peticiones en el puerto indicado por el usuario. Estas peticiones pueden ser:

- Solicitudes de mallado de la geometría para un módulo específico.
- Solicitudes de cálculo de resultados para un módulo especifico.
- Solicitudes de comprobación de disponibilidad del servidor. A estas solicitudes, el servidor debe ser capaz de responder si está ocupado (es decir, está realizando alguna operación de mallado o de cálculo) o si está disponible y listo para aceptar solicitudes.
- Solicitudes para abortar la operación de mallado o de cálculo actualmente en ejecución.

De estas solicitudes, se debe garantizar la exclusividad entre las dos primeras. Es decir, no puede estar ejecutándose más de una operación de mallado o de cálculo en un momento determinado.

Al solicitarse una operación de mallado o de cálculo, el servidor deberá identificar el módulo de la simulación sobre la que se solicita la operación. De esta forma, el servidor podrá invocar al mallador con los parámetros adecuados y llamar al núcleo de simulación asociado al módulo con el que se realiza la simulación. Cuando el servidor termina de realizar la operación solicitada, deberá devolver los resultados al cliente.

3.2 Lista de requisitos del sistema

Una vez descrito el comportamiento del sistema, estamos en condiciones de dar una lista exhaustiva de los requisitos de la aplicación.

[1]Este término no tendría traducción aparente al español, pero se puede interpretar como "servidor de reserva" o "servidor plan B".

La siguiente tabla lista los requisitos de la funcionalidad a desarrollar, así como su prioridad:

Tabla 3.1: Lista de requisitos de la funcionalidad a desarrollar

Identificador	Descripción	Prioridad
RF-PRI-01	El sistema debe permitir al usuario iniciar la aplicación en modo cliente o en modo servidor.	Muy alta
RF-CLI-01	El cliente debe permitir elegir al usuario si desea realizar simulaciones en local o en remoto.	Muy alta
RF-CLI-02	El cliente debe permitir al usuario añadir nuevos servidores sobre los que realizar simulaciones en remoto, especificando dirección IP, puerto y un nombre opcional.	Muy alta
RF-CLI-03	El cliente debe permitir al usuario eliminar servidores añadidos previamente.	Alta
RF-CLI-04	El cliente debe permitir al usuario elegir, de la lista de servidores, el servidor donde desea realizar las simulaciones.	Media
RF-CLI-05	El cliente debe permitir al usuario realizar una operación de comprobación al servidor seleccionado, de forma que éste responda con su estado (desocupado o realizando simulación) o dando un error de falta de conectividad.	Alta
RF-CLI-06	El cliente debe permitir al usuario importar un fichero que contenga información de una lista de servidores.	Baja
RF-CLI-07	El cliente debe permitir al usuario exportar un fichero conteniendo la lista de servidores actual.	Baja
RF-CLI-08	El cliente debe buscar otro servidor en el que realizar una simulación si está activada la opción de *server fallback* y el servidor seleccionado no está disponible.	Media
RF-CLI-09	El cliente debe permitir al usuario abortar una operación de simulación solicitada anteriormente y que aún no haya finalizado.	Alta

Continúa en la siguiente página...

Identificador	Descripción	Prioridad
RF-SIM-01	El sistema debe realizar el proceso de mallado en remoto si se ha seleccionado algún servidor.	Muy alta
RF-SIM-02	El sistema debe realizar el proceso de cálculo de resultados en remoto si se ha seleccionado algún servidor.	Muy alta
RF-SER-01	El servidor debe detectar, de forma automática, el módulo al que corresponde la simulación que se desea realizar.	Muy alta
RF-SER-02	El servidor debe realizar peticiones de mallado recibidas del cliente y tratarlas adecuadamente según el módulo.	Muy alta
RF-SER-03	El servidor debe realizar peticiones de cálculo de resultados recibidas del cliente y tratarlas adecuadamente según el módulo.	Muy alta
RF-SER-04	El servidor debe responder a las peticiones de comprobación indicando su estado actual (desocupado o realizando simulación).	Alta
RF-SER-05	El servidor debe rechazar cualquier petición de simulación (mallado o cálculo) si se encuentra en proceso de realizar otra operación de simulación.	Muy alta
RF-SER-06	El servidor debe abortar la simulación que esté realizando actualmente si recibe una solicitud para abortar la simulación.	Alta
RF-SER-07	El servidor debe devolver los resultados de la operación de mallado o de cálculo al cliente.	Muy alta

Las prioridades de los requisitos indican la importancia de la presencia de dichas características en el producto final. Dar una medida cualitativa de la prioridad a cada requisito nos permite saber a que funcionalidades se les tiene que dar más importancia y por tanto, tienen que ser desarrolladas antes, desarrollando primero las más prioritarias y permitiéndonos dejar aquellas menos prioritarias para una fase más tardía del desarrollo.

Los distintos niveles de prioridad considerados y los criterios seguidos para la calificación de los requisitos se exponen a continuación:

- **Muy alta**: El requisito es indispensable para el correcto desempeño del sistema. En otras palabras, el requisito forma parte de la esencia del proyecto y no puede ser obviado sin que ello resulte en un sistema completamente disfuncional.

- **Alta**: El requisito es importante para el desempeño del sistema o para la experiencia del usuario, pero puede ser obviado sin que ello dé lugar a un sistema completamente no funcional. Se corresponde con características que son muy importantes para la comodidad del usuario, pero no forman parte de la funcionalidad esencial.

- **Media**: El requisito tiene una importancia media. representando una característica que constituye una mejora para el sistema pero no es esencial para su funcionamiento.

- **Baja**: El requisito representa una característica que, si bien puede ser útil para el usuario, su ausencia afectaría mínimamente a la funcionalidad del sistema.

Por otra parte, en las descripciones de los requisitos antes listados, se puede observar que se ha dado una definición muy vaga sobre los procesos de mallado, cálculo de resultados y devolución de resultados. Esto se debe a que el proyecto no se centra en los entresijos de los procesos de mallado y de cálculo, tratándose éstos como cajas negras que reciben una entrada (geometría o parámetros de simulación) y devuelven una salida (malla o resultados de la simulación). Como se verá más adelante, tanto las entradas como las salidas también se tratarán como cajas negras, abstrayéndonos de como se estructuran y qué información contienen. El análisis y comprensión del funcionamiento interno de los procesos de mallado y de cálculo de resultados queda fuera del alcance de este proyecto de fin de grado.

3.3 Descripción de casos de uso

Con el fin de estudiar los requerimientos de la aplicación desde el punto de vista de las interacciones del usuario con el sistema, y como culminación de la fase de análisis de requisitos, estudiaremos los casos de uso del sistema.

En lo que sigue, se tendrá en cuenta un único actor: el usuario, que es el que desea ejecutar una simulación en una máquina remota. En la práctica, no tiene por qué haber solo una persona involucrada: el servidor puede correr a cargo de un responsable de iniciarlo y mantenerlo en ejecución mientras que varios usuarios pueden hacer uso de dicho servidor.

3.3.1 Inicio de la aplicación

La aplicación debe poder iniciarse en modo servidor o en modo cliente. El usuario es el que decide en que modo se ejecuta la aplicación a la hora de iniciarla.

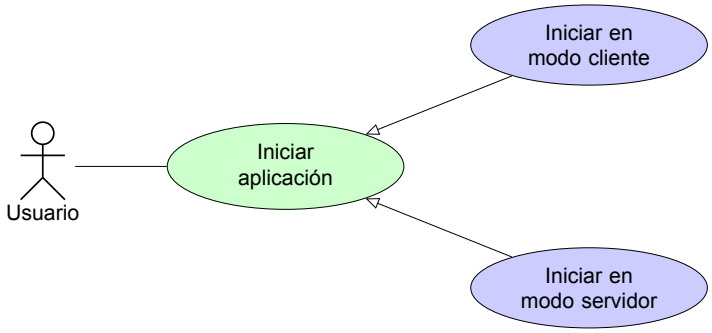

Figura 3.1: Diagrama de casos de uso del inicio de la aplicación

A continuación se da una descripción de los casos de uso mostrados en el diagrama:

Iniciar aplicación

Descripción: Caso de uso abstracto que representa el arranque de la aplicación en cualquiera de los dos modos disponibles (cliente o servidor). El usuario debe decidir en cual de los dos modos desea iniciar la aplicación.

Requisitos involucrados: RF-PRI-01

Iniciar en modo cliente

Descripción: El usuario elige iniciar la aplicación en modo cliente, con lo que la aplicación se iniciará en dicho modo y mostrará una interfaz gráfica de usuario de la misma forma en que lo hacía la aplicación original.

Requisitos involucrados: RF-PRI-01

Iniciar en modo servidor

Descripción: El usuario elige iniciar la aplicación en modo servidor, con lo que la aplicación se iniciará en dicho modo y empezara a escuchar peticiones de simulacion que recibira de los clientes.

Requisitos involucrados: RF-PRI-01

3.3.2 Configuración del remoto

La aplicación cliente debe permitir al usuario configurar la información de los servidores que va a necesitar para realizar sus simulaciones en remoto.

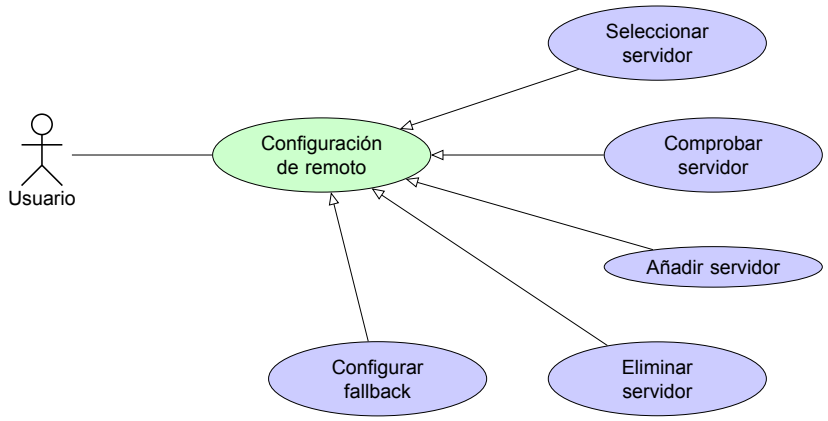

Figura 3.2: Diagrama de casos de uso de la configuración del remoto

A continuación se describen los casos de uso del diagrama anterior:

Configuración del remoto

Descripción: Caso de uso abstracto que representa las distintas posibilidades de configuración de los servidores remotos.

Requisitos involucrados: RF-CLI-01, RF-CLI-02, RF-CLI-03, RF-CLI-04, RF-CLI-05, RF-CLI-06, RF-CLI-07, RF-CLI-08

Descripción de interfaz gráfica: La siguiente interfaz muestra las distintas posibilidades de configuración que ofrece la aplicación:

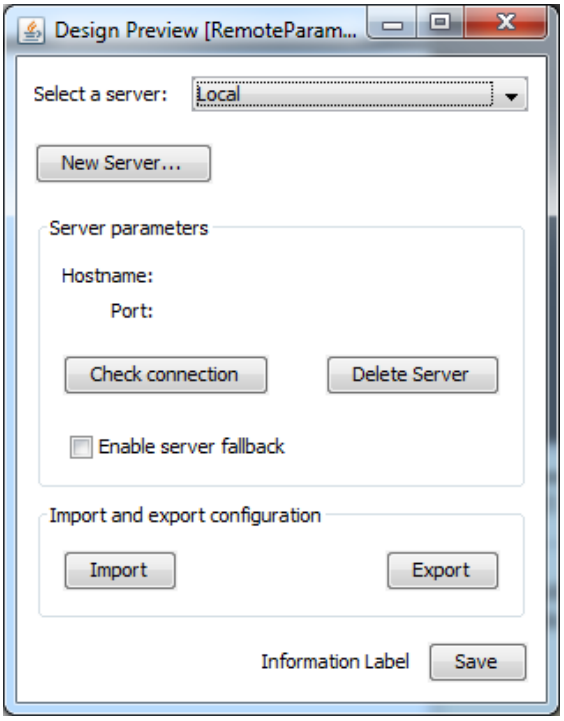

Figura 3.3: Panel de configuración de simulación remota

Añadir servidor

Descripción: El usuario añade un nuevo servidor a la lista de servidores, especificando su dirección de red, número de puerto y, de manera opcional, un nombre identificativo para el servidor.

Postcondición: El servidor nuevo se añade a la lista de servidores.

Requisitos involucrados: RF-CLI-02

Descripción de interfaz gráfica: La siguiente ventana de diálogo es mostrada cuando se desea añadir un nuevo servidor:

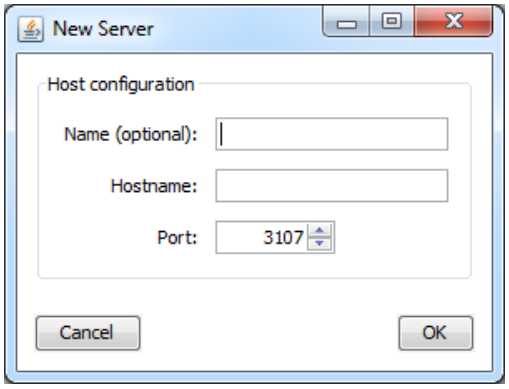

Figura 3.4: Ventana de diálogo para añadir un servidor

Seleccionar servidor

Descripción: El usuario selecciona un servidor de los dados de alta en la lista de servidores, o bien selecciona la opción de realizar las subsecuentes simulaciones de forma local.

Postcondición: El servidor elegido se marca como servidor seleccionado o, en su caso, no se marca ningún servidor como seleccionado (simulaciones locales).

Requisitos involucrados: RF-CLI-01, RF-CLI-04

Eliminar servidor

Descripción: El usuario elimina un servidor dado de alta previamente en la lista de servidores, desechando su información de manera permanente.

Precondición: Debe haber algún servidor de la lista de servidores seleccionado actualmente.

Postcondición: La información del servidor seleccionado se elimina de la lista de servidores.

Requisitos involucrados: RF-CLI-03

Comprobar servidor

Descripción: El usuario realiza una petición de comprobación de disponibilidad al servidor seleccionado para conocer el estado del servidor.

Precondición: Debe haber algún servidor de la lista de servidores seleccionado actualmente.

Requisitos involucrados: RF-CLI-05, RF-SER-04

Configurar fallback

Descripción: El usuario activa o desactiva la opción de "server fallback", que permite a la aplicación cliente probar con otros servidores de la lista si el servidor seleccionado no está disponible.

Precondición: Debe haber algún servidor de la lista de servidores seleccionado actualmente.

Requisitos involucrados: RF-CLI-08

3.3.3 Importación y exportación de configuración

La aplicación cliente debe permitir al usuario importar la configuración de la lista de servidores desde un fichero; o exportar la configuración actual, almacenándola en un fichero.

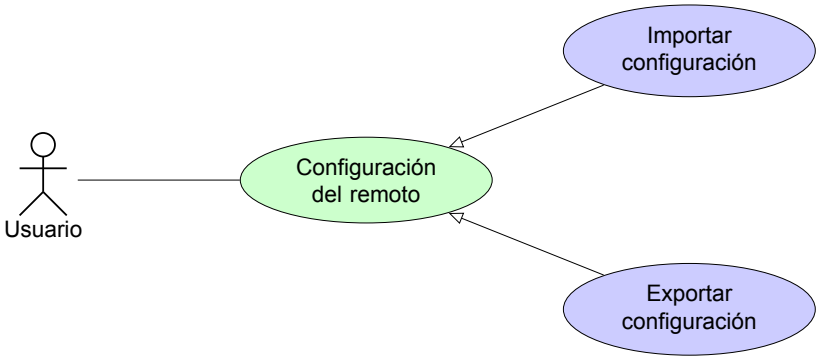

Figura 3.5: Diagrama de casos de uso de la importación y exportación de configuración

Importar configuración

Descripción: El usuario importa una lista de servidores y su configuración (servidor seleccionado y estado de la opción de "server fallback") desde un fichero externo, actualizando la lista de servidores actual de la aplicación.

Precondición: El fichero externo seleccionado debe tener el formato adecuado[2].

Requisitos involucrados: RF-CLI-06

Exportar configuración

Descripción: El usuario exporta la lista de servidores actual y su configuración (servidor seleccionado y estado de la opción de "server fallback"). Esta información se almacenará en un fichero externo, cuya ubicación vendrá dada por el usuario.

Requisitos involucrados: RF-CLI-07

3.3.4 Ejecución de simulaciones remotas

Por último, es necesario tener en cuenta la realización de los procesos de cálculo y de mallado en remoto. El usuario podrá ejecutar estos procesos en remoto si ha configurado algún servidor previamente.

[2]El cual se describirá en el capítulo 4, referente al diseño de la aplicación.

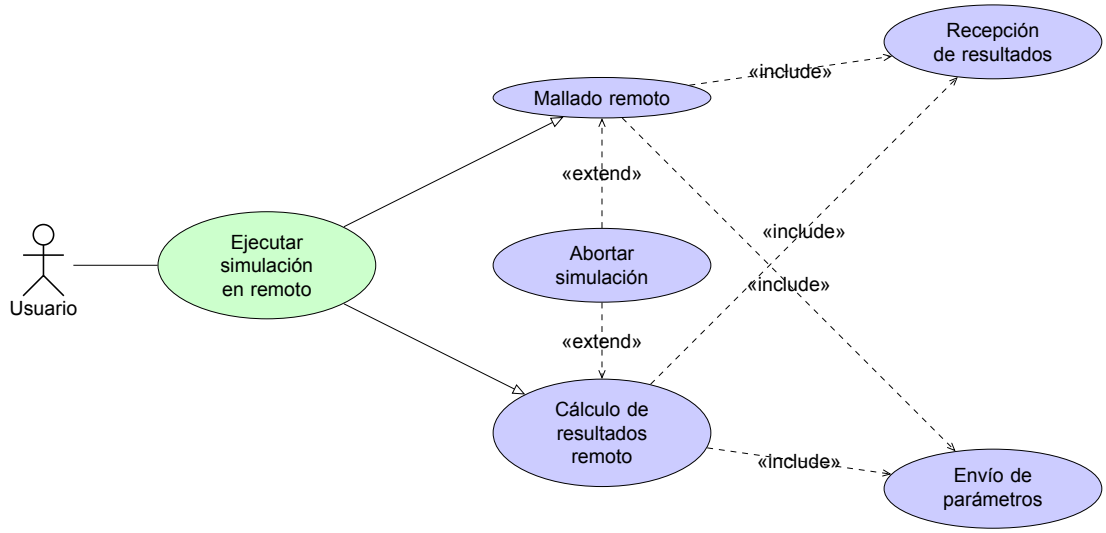

Figura 3.6: Diagrama de casos de uso de la ejecución del mallado y del cálculo

Los casos de uso mostrados en el diagrama de casos de uso y descritos abajo serán comunes a los dos módulos sobre los que se trabaja: MoM y GTD, pues desde el punto de vista del remoto no existe diferencia entre los procesos de los dos módulos, salvo en su funcionamiento interno.

Ejecutar simulación en remoto

Descripción: Caso de uso abstracto. El usuario inicia un proceso de simulación en remoto en el servidor seleccionado.

Precondición: Debe haber algún servidor seleccionado actualmente.

Requisitos involucrados: RF-CLI-09, RF-SIM-01, RF-SIM-02, RF-SER-01, RF-SER-02, RF-SER-03, RF-SER-05, RF-SER-06, RF-SER-07

Mallado remoto

Descripción: El usuario decide realizar una operación de mallado de cualquiera de los dos módulos mencionados en un servidor remoto. Al terminar el mallado remoto, el cliente recibe la malla resultante.

Precondición: Debe haber algún servidor seleccionado actualmente y dicho servidor debe estar disponible (la máquina en que se aloje debe estar accesible, en ejecución y no estar ocupado realizando otra operación).

Requisitos involucrados: RF-SIM-01, RF-SER-01, RF-SER-02

Cálculo de resultados remoto

Descripción: El usuario decide realizar una operación de cálculo de resultados, de cualquiera de los dos módulos mencionados, en un servidor remoto. Al terminar el cálculo remoto, el cliente recibe los resultados de la simulación.

Precondición: Debe haber algún servidor seleccionado actualmente y dicho servidor debe estar disponible.

Requisitos involucrados: RF-SIM-02, RF-SER-01, RF-SER-03

Envío de parámetros

Descripción: El cliente envía los parámetros de la simulación al servidor, el cual los utilizará para alguna operación de simulación. El servidor detectará, a partir de los parámetros recibidos, a qué módulo corresponden.

Requisitos involucrados: RF-SIM-01, RF-SIM-02, RF-SER-01

Recepción de resultados

Descripción: El cliente recibe del servidor los resultados de la operación de simulación (ya sea mallado o cálculo de resultados). El cliente deberá actualizar la información de la simulación actual con los resultados recibidos del servidor.

Requisitos involucrados: RF-SIM-01, RF-SIM-02, RF-SER-07

Abortar simulación

Descripción: El cliente decide abortar una operación de simulación remota que ha iniciado anteriormente. Al abortar la simulación remota, se finalizan las tareas en ejecución en la máquina servidor y se corta la conexión entre el cliente y el servidor, no recibiéndose ningún resultado de la simulación.

Precondición: Debe haber una operación de simulación remota en curso.

Requisitos involucrados: RF-CLI-09, RF-SER-06

Capítulo 4

Diseño

Una vez se ha completado la fase de captura y análisis de requisitos, es momento de pensar en el diseño. El diseño es la fase en la que se decide la arquitectura software del sistema que, dado que este proyecto es un desarrollo orientado a objetos, consistirá en la especificación de las clases del sistema, así como la descripción de las relaciones que existirán entre ellas y los objetos más importantes que existirán durante la ejecución de la aplicación.

En este capítulo, se hablará primero del diseño previo de la aplicación; es decir, de aquellas partes relevantes del diseño original de la aplicación, a modo de introducción para hablar del diseño de la funcionalidad de simulación remota. Posteriormente, se hablará del protocolo de comunicación (protocolo de aplicación) a implementar para que sea posible la interacción entre cliente y servidor, mostrando también la arquitectura software que le dará soporte. Más adelante, nos centraremos en el diseño de los dos componentes del modelo distribuido del proyecto: la parte cliente y la parte servidor.

Durante todo este capítulo, se intentarán justificar las elecciones realizadas en el diseño, apoyándonos en patrones de diseño y otros principios de diseño de la orientación a objetos.

4.1 Diseño previo

En esta sección describemos, de manera breve, aquellas partes del diseño de la aplicación *newFASANT* que se consideren relevantes para el desarrollo de la funcionalidad de simulación en remoto.

4.1.1 `MainFrame` e información de proyecto

El objeto más importante de la aplicación es el conocido como `MainFrame`. Este objeto, que es creado al inicio de la aplicación y permanece durante toda su ejecución,

almacena la información global de la aplicación y representa la ventana principal de la misma. Esto quiere decir que la mayor parte de los elementos de la interfaz gráfica y de la aplicación, como la barra de estado, la información del proyecto actual o el comando actualmente en ejecución, deberán ser accedidos usando este objeto.

En el `MainFrame`, como se ha mencionado, se almacena la información del proyecto en uno de sus atributos. Este atributo es una referencia a la clase `Project`, que es de la cual heredan las distintas clases que representan datos de proyectos de los distintos módulos de la aplicación. Concretamente, `MomData` y `GtdData` son las clases que representan la configuración de los dos módulos con los que se trabaja en este proyecto de fin de grado.

Los objetos de las dos clases mencionadas almacenan la información general del proyecto que contiene la clase `Project`, como la geometría, los materiales y el historial de comandos, así como información específica del propio módulo, como información de parámetros de simulación específicos y datos de antenas (en aquellos módulos que trabajen con antenas).

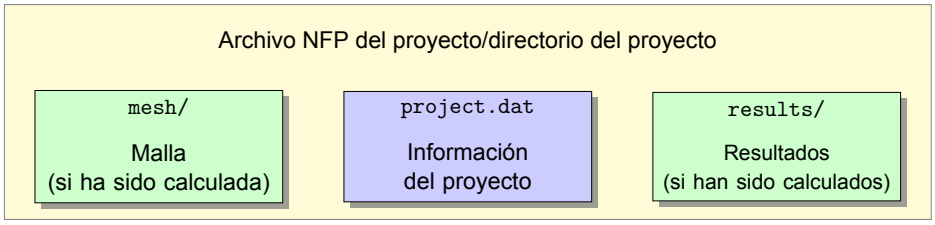

Figura 4.1: Representación gráfica de la estructura del directorio de proyecto

La información contenida en este objeto se serializa en un fichero binario llamado `project.dat` dentro del directorio del proyecto, junto con los resultados de los procesos de mallado y cálculo si han sido ejecutados, en cuyo caso se almacenan en directorios llamados `mesh` y `result` respectivamente. Estos ficheros representan toda la información que contiene un proyecto de la herramienta *newFASANT*. El directorio del proyecto es un directorio cuyo nombre es un UUID aleatorio que, presumiblemente[1], identifica de manera unívoca al proyecto. Este directorio es temporal, en el sentido que solo existe durante el periodo de tiempo en que se tiene abierta la aplicación, y es eliminado cuando se cierra la misma. Por tanto, solo es utilizado temporalmente para guardar la información del mismo de manera no persistente. La estructura de este directorio se puede ver de forma gráfica en la Figura 4.1.

Como cabría esperar, la aplicación es capaz de guardar un proyecto de forma persistente. De esta forma, un usuario sería capaz de guardar el proyecto para distribuirlo o para cargarlo en otro momento. Al guardar un proyecto, la herramienta crea un fichero con extensión NFP (*NewFasant Project*) que en realidad es el directorio del proyecto, con

[1]Un UUID es un número de 128 bits y, por tanto, hay 2^{128} valores diferentes para un UUID, con lo que aunque sea posible generar dos UUID iguales para el mismo proyecto, la probabilidad de que esto ocurra usando un generador pseudoaleatorio es despreciable.

los elementos antes citados, archivado y comprimido en formato 7z. Como se verá en la Sección 4.2, estos ficheros NFP se utilizarán para el intercambio de datos de simulación entre el cliente y el servidor.

Para cargar y guardar ficheros NFP, se dispone de los métodos `loadProject()` y `saveProject()`, respectivamente, de la clase `Project`, los cuales toman la ruta al fichero NFP que se desea cargar en la aplicación como parámetro. El segundo método es crucial para el desarrollo de la funcionalidad de simulación remota, puesto que permite la generación de los ficheros NFP que se intercambiarán entre cliente y servidor. El método `loadProject()` necesita, no obstante, una pequeña variación, puesto que este método hace que se muestre una ventana de diálogo si existe un proyecto cargado en la aplicación previamente (solicitando si deseamos guardar el proyecto anterior). Dado que deseamos que tanto el cliente como el servidor sean capaces de cargar proyectos sin que sea necesaria intervención humana (para responder a la ventana de diálogo), ha sido necesario desarrollar un método `swapProject()` que realice la misma tarea que `loadProject()`, pero sin mostrar dicho mensaje de confirmación[2].

4.1.2 Comunicación con los núcleos de simulación y mallador

La aplicación *newFASANT* consta de una serie de módulos que ofrecen diversas posibilidades de simulación. En el código de la aplicación, cada módulo está organizado en su propio paquete Java, existiendo cuatro subpaquetes en su interior, siendo los más relevantes los dos siguientes:

- `Data`: Contiene las clases que representan la información del proyecto para cada módulo, como parámetros de simulación, resultados, etc. En estos paquetes se encuentran las subclases de la clase `Project`.
- `Frames`: Contiene las clases que representan la interfaz gráfica de usuario del módulo en cuestión. En estos paquetes se encuentran las clases `ExecuteMeshing` y `ExecuteKernel` que se ocupan de la invocación del mallador y del correspondiente núcleo de simulación con los parámetros definidos en la interfaz gráfica y representados por clases del paquete `Data`. Estas dos clases, existentes en cada uno de los paquetes de los módulos, heredan de la clase `ProcessingThread`. Asimismo, en este paquete se encuentran las clases de la interfaz gráfica que se encargan de llamar a las dos clases anteriores, y que son `MeshingParameters` y `CalculateParameters`.

La clase `ProcessingThread` representa la ejecución de un proceso de simulación, ya sea mallado o cálculo. Por otro lado, la clase `ProcessingCommand` representa el panel en el que se muestra un log de proceso (donde se vuelca la salida del núcleo de simulación o

[2]Otra opción más simple, y posiblemente más apropiada, hubiera sido incluir un parámetro booleano en el método `loadProject()` que indicase si mostrar o no dicha ventana de confirmación. Se ha descartado esta opción para seguir conforme al objetivo de modificar la menor cantidad posible de código existente, así como facilitar la integración manual de los cambios ya que el incluir un parámetro al método implicaría cambiar todas sus invocaciones.

mallador) y la opción de abortar. Al lanzar un proceso de simulación, se invoca al método `execute()` del objeto `ProcessingCommand` pasándole como parámetro un objeto de tipo `ProcessingThread`, lo cual inicia el proceso de invocación al kernel y el volcado de su salida al log.

Por comodidad, en la superclase `Project` se han definido dos métodos: `mesh()` y `execute()`, que corresponden a los procesos de mallado y de cálculo de resultados respectivamente. Estos métodos detectan el módulo que corresponde al proyecto sobre el que se invocan y, en función del mismo, devuelven el objeto del tipo `ProcessingThread` que se encarga de llamar, de forma local, al mallador o al núcleo de simulación correspondiente al módulo del proyecto, esto es, un objeto `ExecuteMeshing` o `ExecuteKernel` de los mencionados anteriormente. Este objeto de tipo `ProcessingThread` será el invocado cuando se realice la operación de mallado o cálculo en el lado servidor, y será el objeto que represente la abstracción que ocultará el proceso de mallado o ejecución subyacente, por lo que desde el punto de vista de la funcionalidad remota, estos procesos serán una caja negra.

4.2 Protocolo de comunicación

Con el objetivo de que los módulos cliente y servidor sean capaces de comunicarse entre sí, es necesario establecer un protocolo de aplicación que defina las semánticas, los mensajes intercambiados entre cliente y servidor y su orden. Por ello, primero se definirá una especificación del protocolo y posteriormente se describirá la parte del diseño de la aplicación Java que implementa dicho protocolo.

4.2.1 Especificación del protocolo

El protocolo, como ya se ha mencionado en la Sección 3.3, necesitará soportar dos casos de uso principalmente:

- El usuario deberá ser capaz de comprobar la disponibilidad de un servidor, respondiendo si se encuentra disponible para aceptar nuevas simulaciones o si, por el contrario, se encuentra ocupado y no puede aceptar nuevas solicitudes por el momento.
- El usuario deberá ser capaz de ordenar una nueva operación de simulación a un servidor remoto, enviándole los datos de la simulación y recibiendo los resultados de dicha operación de simulación.

Pasaremos ahora a analizar ambos casos:

Comprobación de conexión

La secuencia de pasos ideada para la realización de una solicitud de conexión es la siguiente:

1. El cliente decide realizar una solicitud de comprobación al servidor seleccionado, abriendo una nueva conexión TCP (bidireccional) con el servidor.
2. El cliente envía un mensaje de tipo `CHECK` para indicar una solicitud de comprobación de conexión.
3. El servidor comprueba si está realizando alguna operación de simulación. De ser así, devuelve al cliente un mensaje de tipo `BUSY`. Si no está realizando ninguna operación de simulación, se devuelve un mensaje de tipo `OK`.
4. Se cierra la conexión entre cliente y servidor.

Esta secuencia de operaciones se describe de manera gráfica en la Figura 4.2.

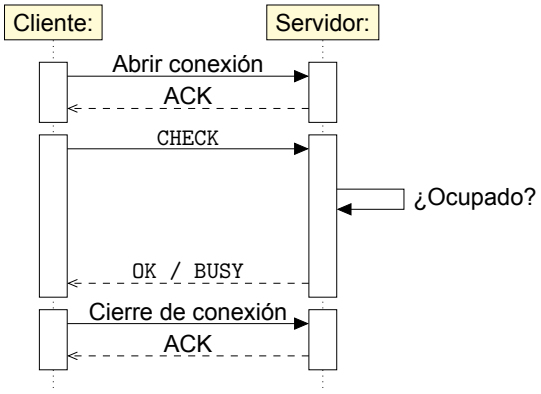

Figura 4.2: Diagrama de secuencia mostrando los pasos para la comprobación del servidor

Solicitud de simulación

La solicitud de simulación deberá comprobar si el servidor en el cual se realiza la simulación está disponible para aceptar peticiones de simulación. Para este fin, podemos reutilizar el sistema de comprobación de conexión descrito, el cual se llevará a cabo previamente a la solicitud de simulación. El proceso completo quedaría como sigue:

1. El cliente y el servidor realizan el proceso de comprobación de conexión, descrito anteriormente.
2. Se abre una conexión TCP para realizar la comunicación.
3. Si el servidor se encuentra disponible; es decir, no está realizando ninguna otra operación de simulación, se envía un mensaje `MESH` o `EXECUTE`, en función de si se trata de una operación de mallado o de cálculo de resultados.

4. El servidor responderá con un mensaje `OK` si sigue libre, en caso contrario, responderá `BUSY` y finalizará el proceso. La recepción, por parte del cliente, de un mensaje `OK` implica que el servidor pasará a estar reservado para ese cliente, pasando a estar ocupado para el resto de clientes hasta que finalice el proceso de simulación del cliente actual.
5. El cliente enviará un mensaje `FILE` con un parámetro `projectSize` indicando el tamaño del proyecto. Posteriormente se enviará el proyecto, en formato NFP, del proyecto sobre el cual se desee realizar la simulación.
6. El servidor recibirá el proyecto, detectará el módulo asociado al proyecto e invocará al núcleo de simulación o mallador, según la operación que se haya solicitado en el paso 3 y el módulo del que proceda el proyecto.
7. Tras la finalización del núcleo o mallador, el servidor enviará un mensaje `FINISH`, y posteriormente un mensaje `FILE` con un parámetro `projectSize` seguido del proyecto resultante en formato NFP, el cual contendrá, salvo excepciones del mallador o núcleo, los resultados calculados.
8. Se cierra la conexión TCP entre cliente y servidor.

En la Figura 4.3 se puede ver un diagrama de secuencia mostrando el proceso anterior, descrito de forma gráfica.

También es necesario tener en cuenta que, mientras se están realizando los pasos 3-7 (ambos inclusive), el usuario puede abortar la operación de simulación que se esté realizando en el servidor, matando a los procesos del núcleo de simulación o mallado que haya ejecutándose en el servidor, así como cortando la conexión entre ambas partes. Para enviar una solicitud de aborción, es necesario enviar un mensaje `ABORT` en una conexión paralela a la utilizada para realizar la solicitud de simulación.

Como se puede ver, a los mensajes intercambiados entre cliente y servidor se les han dado nombres como `FILE`, `MESH` o `EXECUTE`. Estos serán los tipos de mensajes que necesitaremos implementar en la aplicación. Además, deberemos tener en cuenta que los mensajes podrán disponer de parámetros de distinto tipo, como el tamaño del proyecto en los mensajes `FILE`, o el mensaje informativo que llevarán los mensajes resultantes de una comprobación de conexión.

4.2.2 Canales y mensajes remotos

Para hacer posible el protocolo anterior, es necesario diseñar e implementar las clases de la aplicación que representarán a los mensajes, y las que se encargarán de la transmisión de los mensajes. En nuestro caso, disponemos de las clases `RemoteMessage` y `RemoteChannel` que representan, respectivamente, los mensajes del protocolo y una abstracción, construida sobre el mecanismo de los sockets de Java, a través de la cual se trasmiten los datos. Estas clases, al ser necesarias para la comunicación en ambos sentidos, serán mantenidas en un paquete compartido que llamaremos `Shared`. Las clases

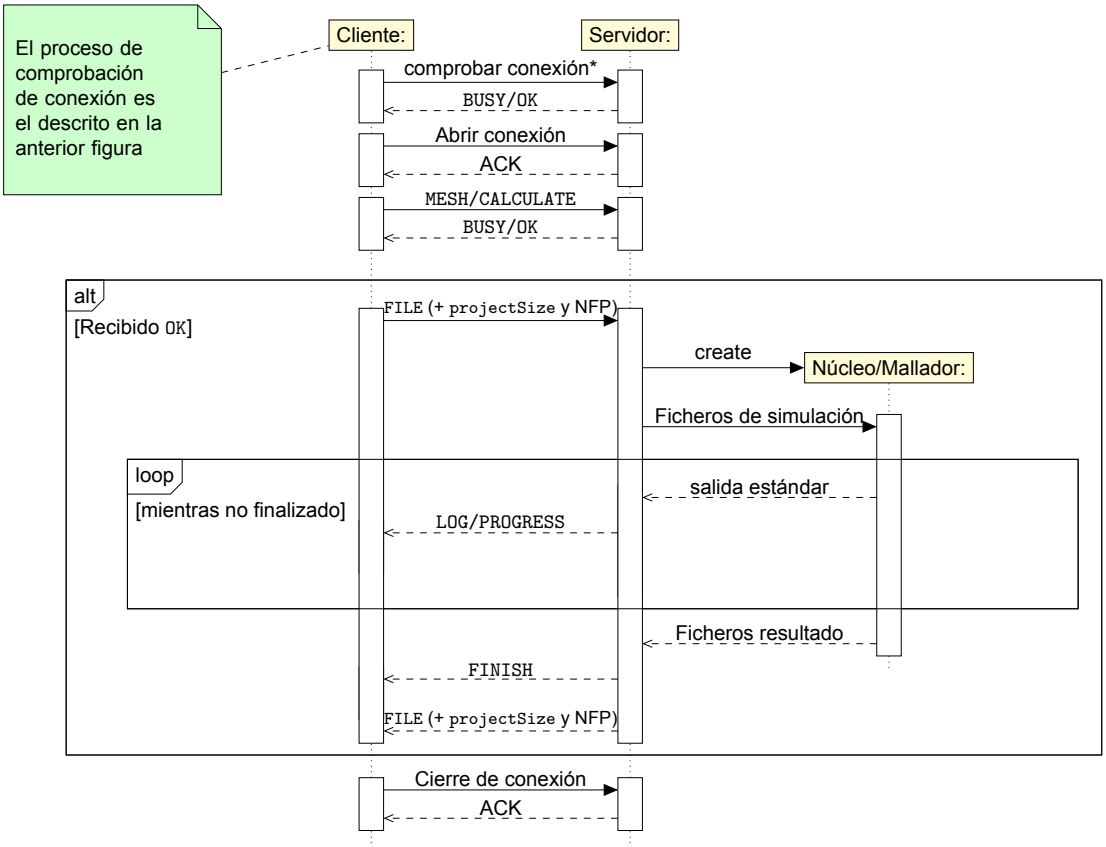

Figura 4.3: Diagrama de secuencia mostrando los pasos para la realización de una simulación remota

que se encuentren en este paquete serán usadas tanto por la parte cliente como por la parte servidor.

En la Figura 4.4 se puede ver un diagrama de clases del paquete `Shared`. A continuación se describirá el rol de cada una de estas clases:

Clase `RemoteMessage`

Representa un mensaje del protocolo de aplicación descrito en la Sección 4.2. Cada mensaje lleva asociado un tipo, representado por el tipo enumerado `Action`, así como un conjunto de pares clave-valor como parámetros que se almacenan en un `Map` de Java.

Los parámetros almacenados en un mensaje remoto se representan internamente como cadenas de caracteres (`String`). No obstante, el cliente de esta clase puede elegir el tipo resultante a la hora de recuperar un parámetro, siendo las posibles elecciones: `String`, `int` o `boolean`. Esto permite al usuario de la clase cierta flexibilidad a la hora

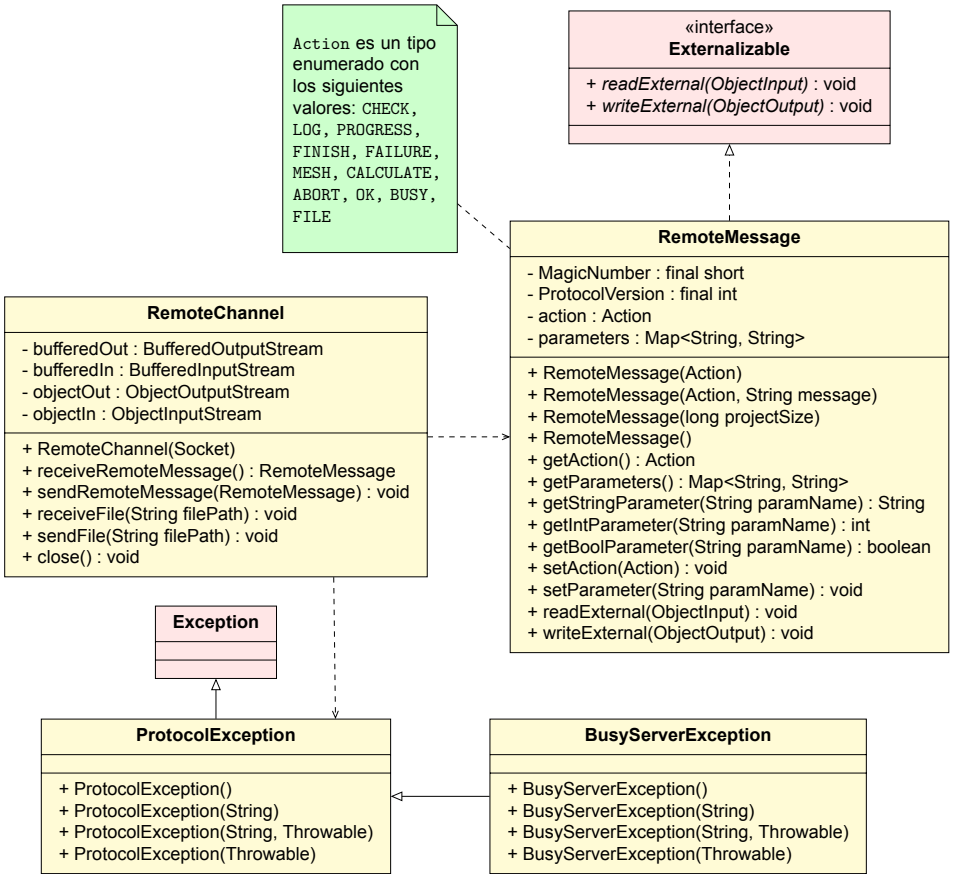

Figura 4.4: Diagrama de clases del paquete `Shared`

de transportar parámetros de diferentes tipos entre cliente y servidor. Para almacenar un parámetro, se tiene que convertir el tipo original a cadena de caracteres previamente (proceso que se hace automáticamente, puesto que Java llama al método `toString()` cuando se requiere una cadena de caracteres).

Por otro lado, la clase `RemoteMessage` implementa la interfaz `Externalizable` del API de Java ya que, como desarrolladores, queremos controlar la forma en la que se serializan los mensajes, en lugar de delegar esta tarea a Java. Haciendo esto, podemos buscar una forma de serializar los mensajes de forma que requieran menos espacio y, por tanto, la sobrecarga en la red sea menor. Se explicará más detalladamente el proceso de serialización y deserialización de mensajes en el Capítulo 5.

Por último, con el fin de controlar la compatibilidad entre las distintas versiones de la aplicación, las cuales pueden tener versiones distintas del protocolo, se tiene el atributo constante `ProtocolVersion`, el cual es un valor numérico que representa la versión del protocolo. Si alguno de los lados, ya sea cliente o servidor, detecta que el otro envía un mensaje con un número de versión distinto de protocolo, ocurrirá una excepción al

deserializar el mensaje, indicando que alguna de las aplicaciones debe ser actualizada. Por tanto, este número de versión debe ser cambiado en el código cuando se haga algún cambio en el protocolo que requiera romper la compatibilidad con versiones anteriores.

Clase `RemoteChannel`

Un objeto de la clase `RemoteChannel` se trata como una abstracción que representa el canal de comunicación entre cliente y servidor, siendo una capa de más alto nivel por encima del mecanismo de los sockets y de entrada/salida de Java. El uso de objetos `RemoteChannel` en el código de cliente y servidor, en lugar de establecer la comunicación directamente usando sockets permite facilitar la mantenibilidad del código, ya que el código encargado de los entresijos de la comunicación se mantiene en una única clase.

Para instanciar la clase `RemoteChannel`, es necesario pasar el objeto `Socket` usado para la conexión TCP entre cliente y servidor. Una vez hecho esto, el `RemoteChannel` creará todos los streams de entrada/salida necesarios para la transmisión y recepción de datos.

Una vez creado, el `RemoteChannel` puede ser usado para el envío y recepción de mensajes remotos (`RemoteMessage`) de cualquier tipo con los métodos `sendRemoteMessage()` y `receiveRemoteMessage()`.

Por último, esta clase también permite el envío y recepción de ficheros con los métodos `receiveFile()` y `sendFile()`, los cuales toman como único parámetro la ruta del fichero en ambos casos. Como ya hemos visto, el envío de un fichero consiste en la transmisión de un mensaje `FILE` con un parámetro `projectSize` y posteriormente, el fichero en sí, por lo que estos métodos ocultarán, de forma transparente, toda esta complejidad.

Clases `ProtocolException` y `BusyServerException`

Aunque se haya dado una especificación del protocolo, y se hayan programado cliente y servidor para seguir dicho protocolo, nunca hay que olvidarse de los casos en los que la comunicación sale mal. Es por ello que se dispone de las excepciones que, como su nombre indica, son objetos que representan la existencia de situaciones excepcionales durante la ejecución del programa.

En nuestro caso, disponemos de la clase `ProtocolException`, la cual hereda de la clase `Exception` del API de Java. Esta excepción es usada en los casos en que la comunicación entre el cliente y el servidor se desvía de la especificación del protocolo, como por ejemplo: tipos de `RemoteMessage` no esperados. La subclase `BusyServerException` representa el caso en que el servidor recibe una solicitud de simulación mientras está realizando otra simulación diferente.

4.3 Diseño del cliente

En esta sección describiremos el diseño de la parte cliente de la aplicación, hablando de las clases utilizadas para almacenar la información de los servidores, para realizar la comprobación remota de un servidor y del diseño de la parte cliente del protocolo.

4.3.1 Configuración del remoto

Empezaremos por ver como se representan las listas de servidores dentro de la aplicación. La arquitectura software (clases y relación entre las mismas) se puede ver en la Figura 4.5. Las clases de las que se hablará a continuación se encuentran en el subpaquete `Client.Config`.

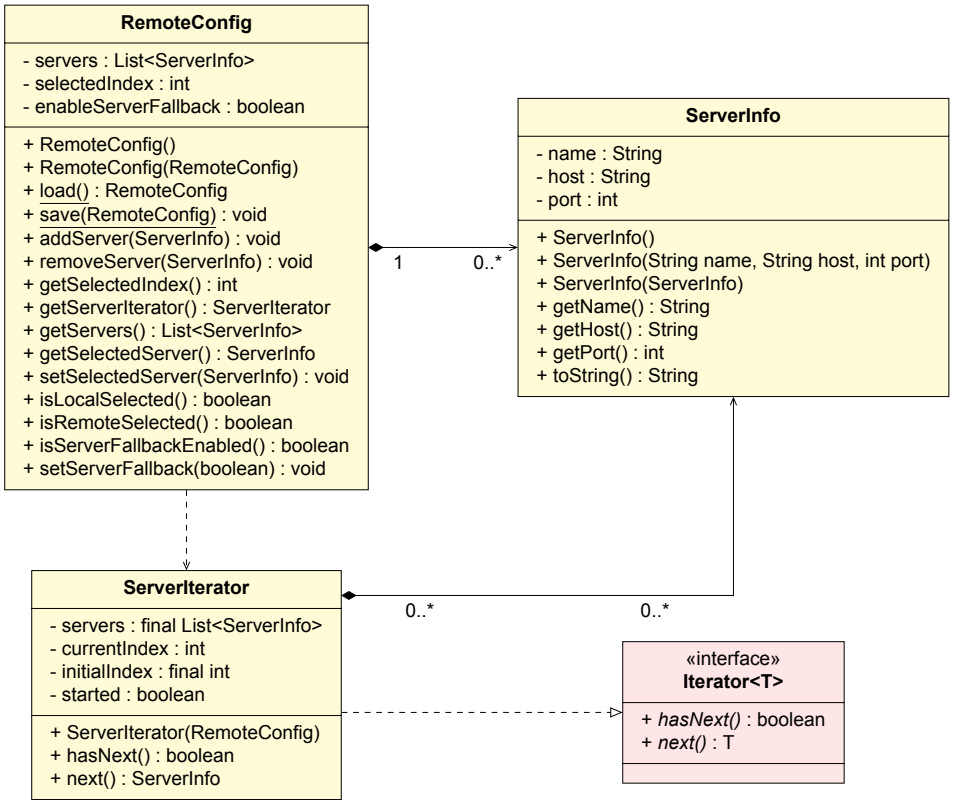

Figura 4.5: Diagrama de clases del paquete `Client.Config`

Clase `ServerInfo`

La clase `ServerInfo` representa a los objetos que almacenan la información de un servidor. Cada objeto mantiene información sobre el nombre de host del servidor (el

cual puede ser un nombre de host o una dirección IP al uso), el número de puerto en el que escucha peticiones el servidor (por defecto, el puerto 3107) y, de forma opcional, un nombre identificativo para la conexión, dado por el usuario.

Como detalle a destacar, se ha sobrecargado el método `toString()` para que devuelva una cadena de caracteres representativa del servidor; esto es, que devuelva el nombre identificativo del servidor si existe y, si no, que devuelva una cadena formada por el nombre de host y el puerto.

Clase `RemoteConfig`

La clase `RemoteConfig` representa la configuración de servidores de la aplicación; esto es, la lista de servidores (objetos de la clase `ServerInfo`) dados de alta. También se distingue entre si hay un servidor seleccionado actualmente (con lo cual, estarían activadas las simulaciones en remoto) o si no hay ningún servidor seleccionado. Además, la clase incorpora todos los métodos necesarios para añadir y eliminar servidores, así como obtener y cambiar el servidor seleccionado actualmente (en caso de simulaciones en local, el servidor seleccionado será `null` y el atributo `selectedIndex` tendrá valor -1). Este objeto también almacenará el estado de la opción de *server fallback*.

Se incluyen dos métodos estáticos: `load()` cargará la configuración remota desde el fichero XML `remote.xml` presente en el mismo directorio que el ejecutable de la aplicación. Este método es llamado al inicio de la aplicación (desde el `MainFrame`) para inicializar el objeto `RemoteConfig` que representará la configuración de servidores de la aplicación. Por otro lado, `save()` hace el procedimiento inverso, guardando el estado del objeto `RemoteConfig` en el fichero XML para su persistencia.

Patrón *Iterator* y clase `ServerIterator`

Cuando se activa la opción de *server fallback*, es necesario recorrer la lista de servidores si el servidor actualmente seleccionado no está disponible. Para ello, es necesario que el código cliente de la clase `RemoteConfig` disponga de una forma de recorrer cíclicamente los servidores de la lista pero sin exponer detalles de implementación de la clase `RemoteConfig` (el código cliente no tiene por qué conocer como se implementa la lista de servidores, ni siquiera el hecho de que el servidor seleccionado se indica con un índice entero en esa lista).

En estos casos, se hace uso de un patrón de comportamiento ampliamente utilizado llamado **Iterator**. Este patrón de diseño consiste en el desarrollo de un objeto llamado *iterador*, el cual permite recorrer otro objeto que hace el papel de colección, y que en este contexto recibe la calificación de *iterable*. El objeto iterador permite el acceso al objeto iterable usando una interfaz uniforme que, en el caso de la interfaz `Iterator` del API de Java, consta de los métodos `hasNext()`, para comprobar si quedan más elementos en la colección, y `next()`, que devuelve el siguiente elemento de la colección y avanza el

cursor del iterador. Se dice que una colección no tiene más elementos desde el punto de vista de la colección cuando el cursor se encuentra al final de la misma.

En nuestro caso, el iterador es la clase `ServerIterator` y el rol de "iterable" corresponde a la clase `RemoteConfig`. La clase `ServerIterator`, no obstante, es un iterador ciertamente peculiar, ya que el primer elemento que se obtiene es el servidor seleccionado, y la lista se recorre como si fuera una lista circular: al llegar al final de la lista, se empieza desde el principio, y el iterador agota sus elementos cuando su recorrido llega de nuevo al servidor inicial, es decir, el seleccionado inicialmente.

Para obtener un `ServerIterator` asociado a un `RemoteConfig`, basta con llamar al método `getServerIterator()` de la clase `RemoteConfig`. Dado que `ServerIterator` implementa la interfaz `Iterator` de Java, se pueden utilizar los métodos `next()` y `hasNext()` descritos anteriormente.

4.3.2 Estructura del fichero XML de la configuración del remoto

Como ya se ha comentado previamente, la configuración del remoto se almacena de forma persistente en un fichero XML, gracias a la infraestructura JAXB de Java. Los detalles de uso de esta biblioteca se verán en el Capítulo 5. No obstante, podemos dar una especificación, en formato XSD[3], del formato XML que debe tener cualquier fichero válido de configuración de remoto:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4    <xs:element name="server" type="serverInfo"/>
5
6    <xs:element name="serverList" type="remoteConfig"/>
7
8    <xs:complexType name="remoteConfig">
9      <xs:sequence>
10       <xs:element name="servers" minOccurs="0">
11         <xs:complexType>
12           <xs:sequence>
13             <xs:element ref="server" minOccurs="0" maxOccurs="unbounded"/>
14           </xs:sequence>
15         </xs:complexType>
16       </xs:element>
17     </xs:sequence>
18     <xs:attribute name="selected" type="xs:int" use="required"/>
19     <xs:attribute name="enableServerFallback" type="xs:boolean" use="required"/>
20   </xs:complexType>
21
22   <xs:complexType name="serverInfo">
23     <xs:sequence>
24       <xs:element name="host" type="xs:string"/>
```

[3]Generada usando la herramienta `schemagen` incluida en el SDK de Java, pasándole como parámetros los ficheros fuente de las clases `RemoteConfig` y `ServerInfo`.

```
25        <xs:element name="port" type="xs:int"/>
26      </xs:sequence>
27      <xs:attribute name="name" type="xs:string"/>
28    </xs:complexType>
29  </xs:schema>
```

Como se puede ver en el fichero anterior, se definen dos elementos XML: por un lado, el principal, `remoteConfig`, que actuará como elemento raíz y por otro, `server`, que contendrá la información de un servidor. El primer tipo de estos elementos se corresponde con la clase de configuración del remoto, `RemoteConfig`, la cual contiene una lista de objetos de la clase `ServerInfo`, clase a la cual está asociado el segundo tipo de elementos XML.

El elemento raíz `remoteConfig` tiene dos atributos: `selected`, que indica la posición dentro de la lista (basada en 0) del servidor seleccionado actualmente, o -1 si no hay ningún servidor seleccionado (por tanto, está seleccionada la simulación en local); y por otro lado, el atributo `enableServerFallback` que indica si está activada la opción de "server fallback". Dentro del elemento raíz se encuentra un elemento `servers` que, a su vez, contiene los elementos `server` de los servidores que componen la lista.

Por último, cada elemento `server` contiene dos elementos a su vez: `host`, que contiene el nombre de host y `port` que contiene el número de puerto. Además, un elemento `server` contiene un atributo opcional `name` con el nombre de la conexión.

A continuación se muestra un ejemplo de fichero XML que muestra todo lo descrito anteriormente:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <serverList selected="1" enableServerFallback="true">
3      <servers>
4          <server>
5              <host>172.22.64.127</host>
6              <port>3107</port>
7          </server>
8          <server name="Máquina virtual">
9              <host>192.168.56.101</host>
10             <port>3107</port>
11         </server>
12     </servers>
13 </serverList>
```

4.3.3 Interfaz gráfica de usuario y comprobación de conexión

A continuación se detalla el diseño de las clases relacionadas con la interfaz gráfica, así como aquellas relacionadas con el mecanismo de comprobación de conexión. Las clases descritas a continuación pertenecen a los subpaquetes `Client.Checker` y `Client.Frames`, y se puede ver su relación en el diagrama de clases de la Figura 4.6.

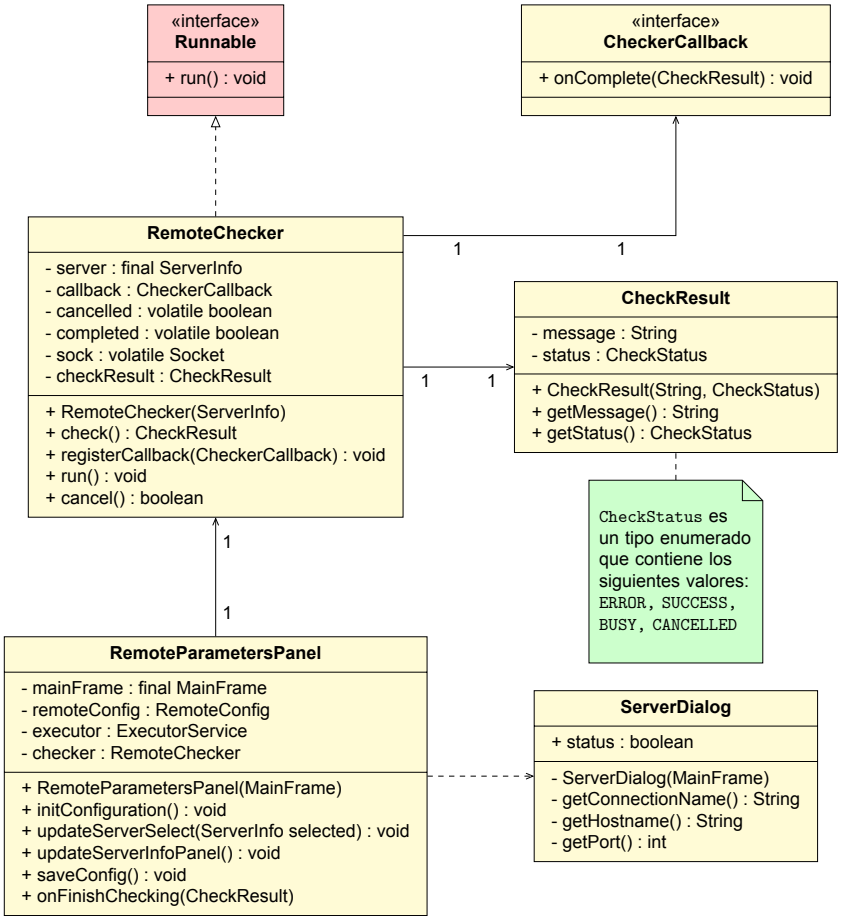

Figura 4.6: Diagrama de clases de los paquetes `Client.Checker` y `Client.Frames`

Clases `RemoteChecker`, `CheckResult` e interfaz `CheckerCallback`

La clase `RemoteChecker` representa una tarea que realiza una comprobación de estado a un servidor remoto. Esta tarea debe poder realizarse de forma síncrona o de forma asíncrona. A continuación se explica en que consiste cada forma:

- **Síncrona**: Se invoca la tarea, y hasta que no finalice, el código llamador no continúa con lo que estaba haciendo al invocarla. Esto es lo que ocurre cuando se realiza la comprobación de servidor previa a la operación de simulación.
- **Asíncrona**: La tarea se realiza en paralelo, y el código llamador continúa su flujo de ejecución. En este caso, es importante establecer una forma de que el código cliente sepa cuándo ha terminado la tarea. La tarea se debe realizar de forma asíncrona cuando es iniciada por el propio usuario en la interfaz gráfica, ya que, de no ser así, la interfaz gráfica se quedaría bloqueada mientras se realiza la comprobación, lo que da una mala sensación al usuario.

El diseño propuesto permite ambas opciones: por un lado, `RemoteChecker` hereda de `Runnable` (interfaz del API de Java), de forma que un objeto de esta clase pueda ser alimentada a un hilo o a un `ExecutorService` para su ejecución de forma asíncrona. En este caso, se permite la asociación de un callback[4] que se ejecutará cuando la comprobación se termine. Este callback estará representado por una implementación de la interfaz `CheckerCallback`, la cual implementará su método `onComplete()` con la acción a realizar al terminar la comprobación. Por otro lado, si se desea invocar a la tarea de comprobación de manera síncrona, basta con llamar al método `run()` de forma directa.

El resultado de una tarea de comprobación se representa mediante un objeto de la clase `CheckResult`, el cual contiene un mensaje procedente del servidor (indicando, de forma textual, si está libre u ocupado) y un valor de un tipo enumerado indicando su estado de manera simbólica, u indicado un error o cancelación de la operación. Una referencia a este objeto `CheckResult` es pasada como parámetro al método `onComplete()` del objeto `CheckerCallback` correspondiente, tras finalizar la comprobación, para que actúe en consecuencia.

Una tarea de tipo `RemoteChecker` debe poder ser cancelada por el usuario en cualquier momento, si se ha ejecutado de forma asíncrona, mediante el método `cancel()`. Esto implica que, potencialmente, el objeto `RemoteChecker` será accedido de forma simultánea por dos hilos de ejecución, lo cual ocasiona problemas en aquellos atributos que sean accedidos por ambos hilos. Los atributos `cancelled`, `completed` y `sock` son esos atributos, por lo que se les ha hecho de tipo `volatile` para evitar problemas de acceso concurrente ya que, de esta forma, los valores de dichas variables siempre serán leídas desde memoria y no habrá valores espurios que puedan causar inconsistencias en las acciones de ambos hilos [15].

Clases `RemoteParametersPanel` y `ServerDialog`

Las clases `RemoteParametersPanel` y `ServerDialog` representan la interfaz gráfica de configuración de simulaciones remotas. Las interfaces gráficas que constituyen estas clases ya han sido mostradas en la Sección 3.3.2 y ahora se describirán con algo más de detalle.

Por un lado, la clase `RemoteParametersPanel` dispone de una copia local de la configuración del remoto de la aplicación (objeto `RemoteConfig`), la cual es modificada directamente por el usuario a través de las opciones proporcionadas en la interfaz gráfica. De esta forma, los cambios solo pueden hacerse persistentes cuando el usuario confirma los cambios realización pulsando el botón "Save". Cuando esto ocurre, el objeto `RemoteConfig` es registrado en el `MainFrame` y se serializan sus contenidos en el fichero

[4] "Callback" es el nombre que suele recibir una función que es invocada al dispararse algún evento de la aplicación, como respuesta a una tarea que se realiza de forma asíncrona (p. ej: pulsación de un botón). En Java, dado que no existen funciones de primer orden (funciones que son objetos per se), un callback se representa como un objeto Java que tiene un método que representa la función callback.

`remote.xml`. De igual forma, cuando el usuario abre la interfaz gráfica, se obtiene el objeto `RemoteConfig` desde el `MainFrame`, se realiza la copia local y se muestran sus datos en la interfaz gráfica.

En la clase `RemoteParametersPanel` se tiene una referencia a un objeto `ExecutorService` que representa un pool de un único hilo. Este pool será el utilizado para la ejecución de las tareas de comprobación remota; es decir, será alimentado con objetos de la clase `RemoteChecker`. Usar un `ExecutorService`, en lugar de un `Thread`, es una solución más limpia puesto que, precisamente, los `ExecutorService` han sido ideados para abstraer al desarrollador del trabajo directo con hilos y le permite centrarse en las tareas (objetos `Runnable`) a realizar. El callback a realizar cuando se realiza la comprobación remota se corresponde con el método `onFinishChecking()`, el cual muestra una ventana de diálogo indicando el estado de la comprobación.

Por otro lado, la clase `ServerDialog` representa la ventana de diálogo para introducir los datos del nuevo servidor. Poco hay que destacar de esta clase, salvo que los datos introducidos se obtienen usando los métodos *getter* y que el atributo `estado` indica si el usuario ha confirmado la inserción del nuevo servidor.

4.3.4 Diseño del protocolo en el lado del cliente

Para terminar con el diseño de la parte cliente de la aplicación, se hablará acerca de las clases que implementan el protocolo de comunicación desde el lado del cliente. Estas clases son las que se encuentran en la raíz del paquete `Client`. En la Figura 4.7 se puede ver el diagrama de clases de esta parte de la aplicación.

Patrones *Template Method* y *Proxy*. Protocolo de cliente

`ExecuteRemote` es una clase abstracta que representa la secuencia de acciones a realizar para la solicitud de una simulación desde el cliente al servidor. Esta clase define una serie de métodos que constituyen una serie de pasos en la realización del protocolo. Dichos métodos son llamados en orden en el método `run()` para llevar a cabo dicha comunicación. La ventaja de realizar varios métodos; uno para cada paso del protocolo, es, además de conseguir una mayor claridad y modularidad del código, la de poder sobrecargar alguno de esos métodos en sus subclases si se desea (1) implementar alguna variación del protocolo o si (2) hay pasos que dependen completamente del tipo de operación de simulación a realizar. En el primero de los casos, la superclase ofrecerá una implementación por defecto del método en cuestión (en nuestro caso, el método `receiveResults()`), mientras que en el segundo, la superclase definirá el método como abstracto y será obligatoria su implementación por las subclases (en nuestro caso, el método `sendData()`). Precisamente en esto consiste el patrón **Template Method**: en que las subclases puedan redefinir como deseen alguno de los pasos de un algoritmo, y donde cada paso es implementado como un método.

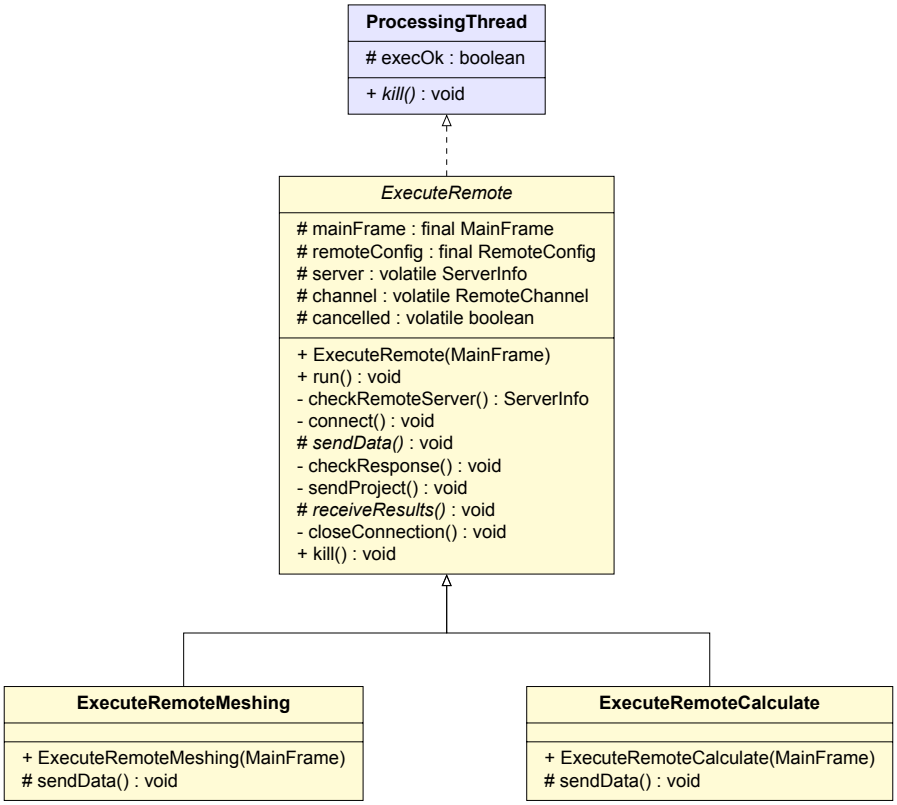

Figura 4.7: Diagrama de las clases encargadas del protocolo de comunicación desde el lado del cliente

Cada uno de los pasos del protocolo corresponden a cada uno de estos métodos (en orden):

1. `checkRemoteServer()`: Comprueba la disponibilidad del servidor seleccionado y, si no está disponible, itera sobre la lista hasta que encuentra alguno disponible. Tira una `ProtocolException` si no hay ninguno disponible o devuelve el primer `ServerInfo` disponible.
2. `connect()`: Establece una conexión con el servidor obtenido del paso anterior, y crea el canal de comunicación.
3. `sendData()`: Envía los metadatos necesarios de la simulación a realizar. Como mínimo se envía un mensaje remoto del tipo de operación que se desea realizar (`MESH` o `EXECUTE`). Este método debe ser implementado por las subclases.
4. `checkResponse()`: Comprueba la respuesta del servidor, esto es, si está libre u ocupado. Si está ocupado, ocurre una `ProtocolException`.
5. `sendProject()`: Guarda el proyecto actualmente cargado en la aplicación cliente y lo envía al servidor, en formato NFP.

6. `receiveResults()`: Recibe las líneas de log del servidor, así como los mensajes de progreso (esto se explicará en la Sección 4.4). Cuando termina la simulación en el servidor, se recibe el proyecto en formato NFP desde el servidor, y el proyecto se carga en la interfaz gráfica.
7. `closeConnection()`: Corta la conexión con el servidor, cerrando el canal y el socket subyacente.

Por otro lado, el método `kill()` es el invocado cuando el botón "Abort" de la interfaz gráfica (clase `ProcessingCommand`) es presionado. Este método abre un canal parelelo con el servidor y envía un mensaje de tipo `ABORT` solicitando la cancelación de la simulación. Dado que este método accede a algunos atributos a la vez que el hilo principal (el que realiza el protocolo), se han definido estos atributos como `volatile`.

Las clases `ExecuteRemoteMeshing` y `ExecuteRemoteCalculate` implementan el método `sendData()` de forma que se envía el comando adecuado al servidor (`MESH` o `CALCULATE`, respectivamente). Estas clases son instanciadas desde las clases `MeshingParameters` o `CalculateParameters` de los módulos MoM y GTD en el caso en el que la opción de simulación remota esté seleccionada. En caso contrario, se utilizarán las clases `ExecuteMeshing` o `ExecuteKernel`, que realizarán las mismas operaciones de forma local. Nótese que, tanto `ExecuteRemoteMeshing` como `ExecuteRemoteCalculate` actúan como intermediarios entre la interfaz gráfica y el servidor, y lo hacen de forma transparente, ya que se ocultan los detalles de la comunicación con el servidor a la interfaz gráfica. Es decir, estas dos clases actúan de la misma forma que si la simulación fuera local gracias a la interfaz que provee la clase `ProcessingThread`. Es por ello que, tanto `ExecuteRemoteMeshing` como `ExecuteRemoteCalculate` son un ejemplo del patrón **Proxy**, ya que actúan en representación del servidor frente al resto de la aplicación cliente.

4.4 Diseño del servidor

Tras hablar del diseño de la parte cliente de la aplicación, se hará lo propio con la parte del servidor. Primero se hablará del diseño de la parte encargada del protocolo y después se describirá la forma en la que el servidor maneja las solicitudes. En la Figura 4.8 se puede observar el diagrama de clases de esta parte de la aplicación.

Es importante notar que el módulo servidor es una aplicación sin interfaz gráfica más alla de la ventana de la consola donde aparecen la salida estándar de la aplicación. Esto es así porque disponer de interfaz gráfica en la aplicación servidor no es necesario ya que la interfaz gráfica que se va a utilizar es la de los módulos cliente, de forma remota. No obstante, la aplicación servidor sigue pudiendo acceder al `MainFrame` y a todos los elementos contenidos en él, teniendo la capacidad de cargar proyectos en la aplicación e, incluso, de acceder a elementos de la interfaz gráfica (la cual está oculta). Este último punto tendrá importancia más adelante.

4.4.1 Diseño del protocolo en el lado del servidor

Las piedras angulares de la implementación del protocolo en el lado servidor consisten en la interfaz `RequestHandler` y sus implementaciones. En este contexto, un *handler* o manejador es un objeto que se encarga de manejar solicitudes del cliente de un tipo concreto. A continuación se explica la función de cada tipo de manejador:

- `CheckRequestHandler`: Se encarga de responder a solicitudes de comprobación de estado, indicando al cliente si el servidor se encuentra o no disponible.
- `BusyRequestHandler`: Se encarga de responder a solicitudes de simulación cuando el servidor se encuentra ocupado y, por tanto, no puede hacer efectiva dicha solicitud. En este caso, se envía un mensaje `BUSY` indicando la no disponibilidad del servidor.
- `SimulationRequestHandler`: Se encarga de responder a solicitudes de simulación. Esta clase es la que tiene la responsabilidad de cargar el proyecto recibido del cliente en la aplicación e invocar al núcleo de simulación o mallador, según el valor del atributo `action` (`MESH` o `CALCULATE`), haciendo uso de los métodos `mesh()` o `execute()` de la clase `Project` y que ya han sido comentados en la Sección 4.1. Tras terminar la simulación, se envía el proyecto resultante al cliente (en formato NFP).
- `AbortHandler`: Se encarga de recibir peticiones de aborción de la simulación actualmente en ejecución. Toma como parámetro el `ServerThread` de la simulación en ejecución y que se debe detener.

Las acciones de las que se encarga cada manejador se implementan en el método `handle()`, mientras que el método `isExclusive()` devuelve si el manejador es exclusivo. Esto último tiene efecto en lo siguiente: dos o más manejadores exclusivos no pueden estar en ejecución a la vez en el servidor, pero sí se pueden combinar un manejador exclusivo y otro que no lo sea. De momento, el único manejador exclusivo es `SimulationRequestHandler`, ya que no se permite la ejecución de varias operaciones de simulación en paralelo.

Por otra parte, la clase `ServerLogger` es usada por `SimulationRequestHandler` con el fin de capturar mensajes de log del servidor y enviárselos al cliente. Antes se ha comentado que, desde el servidor, es posible acceder a los elementos de la interfaz gráfica aunque ésta no sea visible. Es más, mientras se realiza una simulación, los mensajes de log son escritos en el log de proceso (un `JTextArea`) del servidor y la barra de progreso es actualizada con el porcentaje de los cálculos que ha sido realizado. No obstante, se desea que ésta información sea visible en el cliente y, por tanto, es necesario obtenerla de alguna forma para poder enviársela (usando mensajes `PROGRESS` y `LOG`).

Es aquí donde entran los **listeners**. Por un lado, la interfaz `DocumentListener` representa a un listener que escucha eventos de un `JTextArea`, mientras que la interfaz `ChangeListener` se utilizará para escuchar eventos de una `JProgressBar`. Dado que `ServerLogger` implementa ambas interfaces, puede usarse para escuchar eventos de los

dos elementos. De esta forma, se asocia el objeto `ServerLogger` tanto al `JTextArea` como al `JProgressBar` del servidor y, en los métodos `insertUpdate()` y `stateChanged()` se realiza el envío de los mensajes `LOG` y `PROGRESS` con cada línea del log o actualización de la barra de progreso, respectivamente, en respuesta a los eventos que generan los dos elementos de la interfaz gráfica. El cliente obtendrá esta información y la mostrará en su propia interfaz gráfica.

4.4.2 Tratamiento de solicitudes

La clase `Server` es el punto de entrada de la aplicación servidor, instanciada cuando en la clase `Main` se detecta el parámetro `-server` de la línea de comandos. El objeto de la clase `Server`, en su método `startServer()`, se quedará en un bucle de aceptación de conexiones desde el exterior. Tras llegar una conexión, el `Server` creará una instancia de la clase `ServerThread`, el cual se encargará de atender la conexión en paralelo.

El objeto `ServerThread` obtiene, en primer lugar, la petición del cliente y solicitará al `Server` el `RequestHandler` adecuado para llevar a cabo dicha petición, usando su método `acquireRequestHandler()`. La implementación de éste método es bastante delicada, pues si no se protege de la forma adecuada, puede ocurrir que dos `ServerThread` distintos obtengan dos manejadores incompatibles entre sí (esto es, dos manejadores de simulación), rompiendo la regla de que dos manejadores exclusivos actúen a la vez. Es por este motivo que este método es *sincronizado*, de forma que se ejecuta siempre en exclusión mutua y no puedan ocurrir condiciones de carrera que asignen varios manejadores exclusivos a diferentes `ServerThread`. Si un `ServerThread` solicita un manejador de simulación cuando otro ya ha sido asignado, se obtendrá un `BusyRequestHandler` en su lugar.

Por último, cuando el `ServerThread` ha ejecutado el manejador correspondiente, lo liberará invocando al método `releaseRequestHandler()` del objeto `Server`. De esta forma, si el manejador liberado era exclusivo, se dejará paso a una nueva petición que haga uso de otro manejador exclusivo.

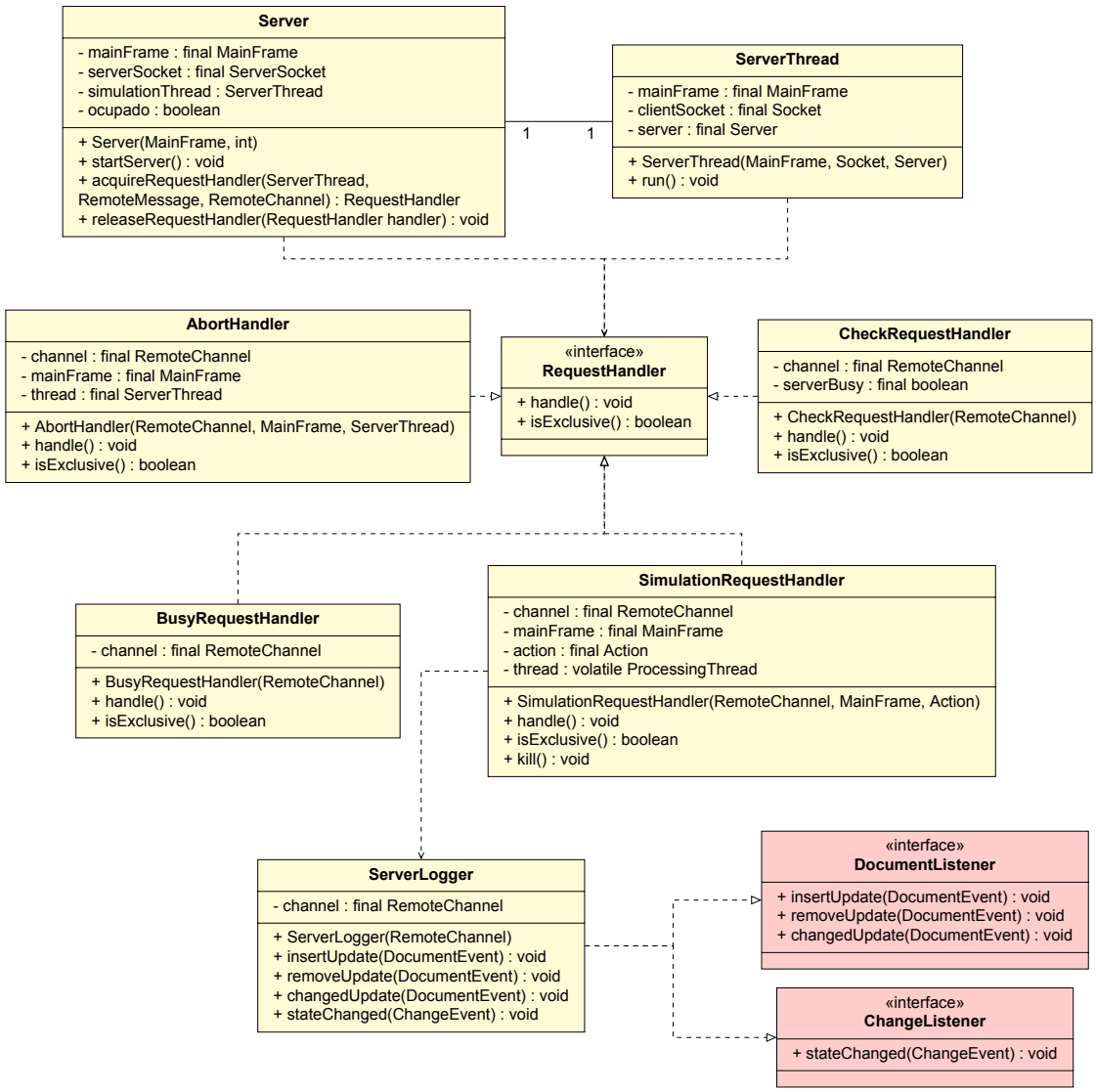

Figura 4.8: Diagrama de clases de la parte servidor de la aplicación

Capítulo 5

Implementación

En este capítulo se destacarán algunos detalles de implementación de la aplicación. Concretamente, se hablará del uso de la biblioteca JAXB para la serialización y deserialización de XML y después, se describirá la forma de serializar los mensajes del protocolo de comunicación.

5.1 Uso de la biblioteca JAXB

5.1.1 Anotaciones Java

JAXB se basa en el uso de las anotaciones de clases y atributos para determinar qué campos de una clase se tienen que serializar y de qué forma. Las anotaciones de Java son pequeños fragmentos de código, encabezados con el carácter "`@`" que ofrecen cierta metainformación sobre clases, métodos y atributos a otras partes de la aplicación; en este caso, a JAXB. Así pues, hay varias anotaciones **de clase** que ofrece JAXB al desarrollador:

- `XmlRootElement`: Permite especificar el nombre del elemento raíz XML en el que se serializará el objeto.
- `XmlType`: Permite especificar el orden de los atributos dentro del elemento XML.
- `XmlAccessorType`: Permite especificar si los valores a serializar provienen de los atributos de la clase o de los métodos getter de la misma. En nuestro caso, siempre se ha seleccionado la primera opción (valor `XmlAccessType.FIELD`).

Por otro lado, hay varias anotaciones **de atributo** que ofrece JAXB:

- `XmlElement`: Permite indicar el nombre del elemento donde se almacena el valor del atributo, así como si es obligatoria su presencia o no.
- `XmlAttribute`: Indica que el atributo (de la clase) se va a tratar como un atributo XML. Permite especificar su nombre y si es, o no, obligatorio.

- `XmlElementWrapper`: Se utiliza en atributos que representan colecciones de otros elementos. Permite especificar el nombre del elemento XML que contiene a los elementos de la colección.

Una característica muy útil de JAXB es su soporte de atributos complejos; es decir, de atributos que no corresponden a tipos básicos sino a otras clases complejas. La clase `RemoteConfig` dispone, además de sus propios atributos, de una lista de objetos de tipo `ServerInfo` que representa la lista de servidores. En este caso, JAXB utiliza las anotaciones de la clase `RemoteConfig` para saber cómo tiene que serializar los atributos de esta clase y, cuando tiene que serializar la lista de objetos `ServerInfo`, utiliza las anotaciones de la clase `ServerInfo` para saber cómo debe guardar sus atributos. El resultado es un único fichero XML con los datos de la clase "contenedora" `RemoteConfig` y los datos de la clase "contenida" `ServerInfo`.

5.1.2 Proceso de serialización y deserialización

La forma de serializar y deserializar objetos XML es bastante sencilla gracias a los objetos `Marshaller` y `Unmarshaller` del API de JAXB. La deserialización del fichero XML se realiza al inicio de la aplicación, mientras que la serialización se realiza cuando el usuario ordena el guardado del archivo presionando el botón "Save" de la interfaz gráfica. En ambos casos, los pasos a realizar son similares:

1. Se obtiene el `JAXBContext` asociado a la clase a serializar/deserializar (en nuestro caso, `RemoteConfig`).
2. Se solicita un objeto `Marshaller` o `Unmarshaller` al contexto creado en el paso anterior.
3. Se invoca el método `marshal()` o `unmarshal()` del objeto `Marshaller` o `Unmarshaller`. En ambos casos se debe especificar el fichero XML utilizado, ya sea de destino o de origen.

Estos pasos se implementan en los métodos estáticos `load()` y `save()` de la clase `RemoteConfig`.

5.2 Serialización de mensajes

Java ofrece la interfaz `Serializable` como mecanismo de serialización de objetos de forma binaria (a diferencia de XML, que es un formato en modo texto). Al hacer que una clase implemente la interfaz `Serializable`, la cual es una interfaz sin métodos, se permite la serialización de los objetos de dicha clase de forma automática; es decir, Java se ocupa íntegramente del formato binario del objeto serializado.

Este mecanismo de Java ofrece una clara ventaja: el desarrollador no tiene que preocuparse por los detalles internos de la serialización ni de la deserialización, pues

Java se ocupa de realizar ambos procesos de forma transparente al programador. No obstante, este mismo punto se podría utilizar, en algunos casos, como una desventaja.

Es por este motivo que Java ofrece otra forma de manejar la serialización y dar más control al desarrollador sobre como ésta se realiza: la interfaz `Externalizable`, que podrá ser implementada por la clase que se desee serializar. Esta interfaz define dos métodos: `writeExternal()` y `readExternal()`, los cuales se ocupan de la serialización y deserialización.

La clase `RemoteMessage` implementa la interfaz `Externalizable` y, por tanto, los dos métodos citados anteriormente. En cuanto a la forma de serializar los mensajes, se escribe la siguiente información:

- Número mágico (`0xFA5A`) que identifica al mensaje como un mensaje del protocolo.
- Versión del protocolo soportada por la aplicación.
- Cadena textual del tipo de mensaje (p. ej: `MESH` o `CHECK`).
- Número de parámetros incluidos en el mensaje. Para cada parámetro, se escribe la siguiente información (en formato de cadena de caracteres):
  - Nombre del parámetro.
  - Valor del parámetro.

A estos datos les acompaña una cabecera introducida por Java al usar un `ObjectOutputStream` (utilizada para almacenar metadatos del objeto, como el nombre de la clase y el paquete en que se encuentra). No obstante, el tamaño del mensaje es significativamente menor que si se usará solamente la interfaz `Serializable`. En las Figuras 5.1 y 5.2 podemos ver dos volcados hexadecimales de un mensaje de tipo `CHECK` (sin parámetros) usando el "método Serializable" y el 'método Externalizable"[1], respectivamente, y se puede ver que la diferencia en tamaño es significativa.

En concreto, se ve que usando el primer método, se tiene un mensaje de 0x14B = 331 bytes, mientras que con el segundo se tiene un mensaje de 0x44 = 68 bytes, consiguiéndose una reducción de la carga de datos de un 79,45 %.

Por último, a la hora de deserializar un mensaje remoto, se hace especial hincapié en que el número mágico y el número de versión del protocolo del mensaje recibido coincida con el número mágico y versión de la aplicación receptora. En caso de no ser así, se lanzará una excepción con un mensaje de error indicando que una de las partes de la aplicación debe ser actualizada.

[1] Estos volcados han sido obtenidos gracias a las herramientas `xxd`, `nc` (netcat) y `hexdump`, presentes en la mayoría de sistemas UNIX. Los comandos `echo "aced0005" | xxd -r -p` sacan por la salida estándar el "número mágico" que espera Java para establecer una conexión cuando usa un `ObjectOutputStream`. De esta forma, se ha hecho que la aplicación envíe un mensaje `CHECK` a netcat, haciéndole pensar que es una aplicación servidor Java normal y corriente. `hexdump` se encarga de mostrar el volcado hexadecimal.

```
jose@vitamin:~/Proyectos/tfg|master⚡
⇒ echo "aced0005" | xxd -r -p | nc -l 3107 | hexdump -C
00000000  ac ed 00 05 73 72 00 1b  52 65 6d 6f 74 65 2e 53  |....sr..Remote.S|
00000010  68 61 72 65 64 2e 52 65  6d 6f 74 65 4d 65 73 73  |hared.RemoteMess|
00000020  61 67 65 dc a9 df 86 f2  ec 77 b1 02 00 04 53 00  |age......w....S.|
00000030  0b 4d 61 67 69 63 4e 75  6d 62 65 72 49 00 0f 50  |.MagicNumberI..P|
00000040  72 6f 74 6f 63 6f 6c 56  65 72 73 69 6f 6e 4c 00  |rotocolVersionL.|
00000050  06 61 63 74 69 6f 6e 74  00 24 4c 52 65 6d 6f 74  |.actiont.$LRemot|
00000060  65 2f 53 68 61 72 65 64  2f 52 65 6d 6f 74 65 4d  |e/Shared/RemoteM|
00000070  65 73 73 61 67 65 24 41  63 74 69 6f 6e 3b 4c 00  |essage$Action;L.|
00000080  0a 70 61 72 61 6d 65 74  65 72 73 74 00 0f 4c 6a  |.parameterst..Lj|
00000090  61 76 61 2f 75 74 69 6c  2f 4d 61 70 3b 78 70 fa  |ava/util/Map;xp.|
000000a0  5a 00 00 00 01 7e 72 00  22 52 65 6d 6f 74 65 2e  |Z....~r."Remote.|
000000b0  53 68 61 72 65 64 2e 52  65 6d 6f 74 65 4d 65 73  |Shared.RemoteMes|
000000c0  73 61 67 65 24 41 63 74  69 6f 6e 00 00 00 00 00  |sage$Action.....|
000000d0  00 00 00 12 00 00 78 72  00 0e 6a 61 76 61 2e 6c  |......xr..java.l|
000000e0  61 6e 67 2e 45 6e 75 6d  00 00 00 00 00 00 00 00  |ang.Enum........|
000000f0  12 00 00 78 70 74 00 05  43 48 45 43 4b 73 72 00  |...xpt..CHECKsr.|
00000100  11 6a 61 76 61 2e 75 74  69 6c 2e 48 61 73 68 4d  |.java.util.HashM|
00000110  61 70 05 07 da c1 c3 16  60 d1 03 00 02 46 00 0a  |ap......`....F..|
00000120  6c 6f 61 64 46 61 63 74  6f 72 49 00 09 74 68 72  |loadFactorI..thr|
00000130  65 73 68 6f 6c 64 78 70  3f 40 00 00 00 00 00 00  |esholdxp?@......|
00000140  77 08 00 00 00 10 00 00  00 00 78                 |w.........x|
0000014b
```

Figura 5.1: Volcado hexadecimal/ASCII de un mensaje `CHECK` usando `Serializable`

```
jose@vitamin:~/Proyectos/tfg|master⚡
⇒ echo "aced0005" | xxd -r -p | nc -l 3107 | hexdump -C
00000000  ac ed 00 05 73 72 00 1b  52 65 6d 6f 74 65 2e 53  |....sr..Remote.S|
00000010  68 61 72 65 64 2e 52 65  6d 6f 74 65 4d 65 73 73  |hared.RemoteMess|
00000020  61 67 65 7a d6 b8 b7 d0  89 94 d7 0c 00 00 78 70  |agez..........xp|
00000030  77 11 fa 5a 00 00 00 01  00 05 43 48 45 43 4b 00  |w..Z......CHECK.|
00000040  00 00 00 78                                       |...x|
00000044
```

Figura 5.2: Volcado hexadecimal/ASCII de un mensaje `CHECK` usando `Externalizable`

Capítulo 6

Caso de prueba

En este capítulo se mostrará un ejemplo de uso de la funcionalidad desarrollada. En este ejemplo, se hará uso de dos equipos con Windows 7; uno funcionando en modo servidor y otro en modo cliente, y se tratará de hacer una simulación en remoto usando el módulo MoM de la herramienta.

6.1 Realización de una simulación en el módulo MoM

Requerimientos previos. Para poder ejecutar este caso de prueba, es necesario disponer de la aplicación *newFASANT* en ambos equipos, siendo necesario disponer de una licencia para la aplicación servidor.

Para mayor comodidad, en sistemas Windows conviene crear un acceso directo a la aplicación que la ejecute en modo servidor, incluyendo como parámetros de la línea de comandos "`-server <puerto>`", donde "`<puerto>`" indica el puerto deseado. Si se desea usar el puerto por defecto (3107), basta con especificar solo el parámetro "`-server`". En sistemas UNIX habría que realizar un pequeño script de la shell para conseguir el mismo objetivo. Se darán instrucciones más detalladas sobre como realizar esto en el Apéndice C.

Paso 1. Ejecutar la aplicación en modo servidor en la máquina que actúe como servidor. Para esto se puede hacer uso del acceso directo o script ya mencionado, o ejecutar el fichero ejecutable directamente desde la línea de comandos pasándole los parámetros necesarios. Para ejecutarlo desde la línea de comandos, es necesario navegar al directorio de instalación de la aplicación y, posteriormente, llamar al ejecutable con el parámetro `-server`. Esto se haría introduciendo los comandos de la siguiente manera:

```
cd <Ruta de instalación>
NewFasant6.exe -server <puerto>
```

Donde `<Ruta de instalación>` se debe sustituir por el directorio en el cual se ha instalado el programa, y `<puerto>` es un número de puerto disponible (si no se especifica, se usará el puerto 3107).

Una vez hecho esto, es posible que aparezca la siguiente ventana, especialmente si es la primera vez que se ejecuta el servidor en el ordenador:

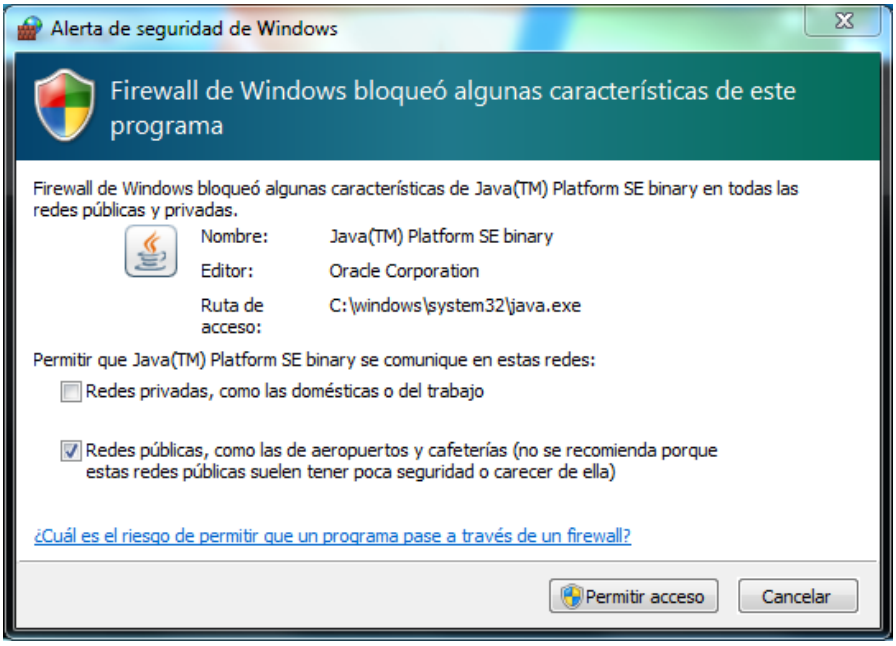

Figura 6.1: Firewall de Windows preguntando si queremos permitir acceso a la red por parte de la aplicación

En este caso, es necesario pulsar el botón “Permitir acceso” y responder afirmativamente a la ventana emergente que aparecerá después. De esta forma, el firewall de Windows hará una excepción y permitirá al servidor escuchar peticiones.

Una vez arrancado el servidor, aparecerá la siguiente ventana de consola indicando que, efectivamente, el servidor se encuentra escuchando en el puerto indicado:

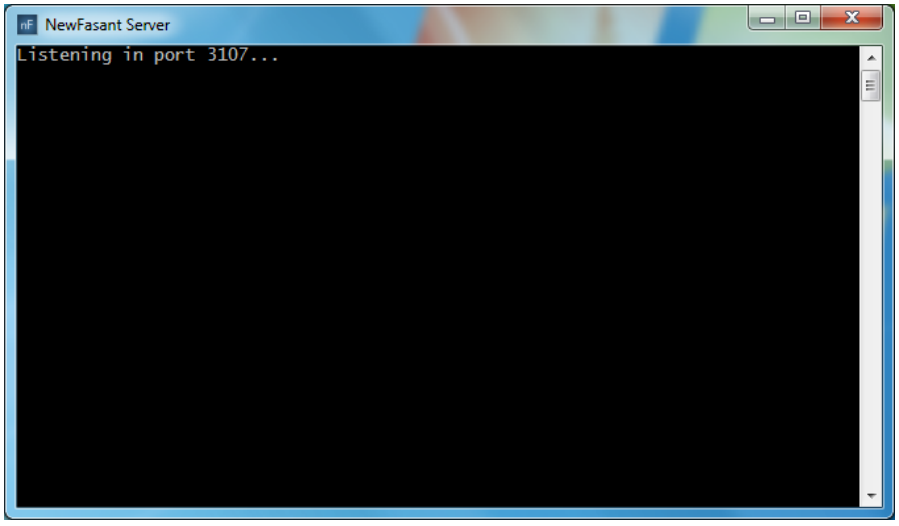

Figura 6.2: Servidor en ejecución

Paso 2. Una vez se ha iniciado el servidor, es posible ejecutar simulaciones en remoto. Para ello, se abre la aplicación *newFASANT* en el cliente (esta vez de la forma normal, sin pasar ningún parámetro de línea de comandos). Una vez abierto, se crea un nuevo proyecto y se selecciona el módulo "MoM" haciendo clic en el siguiente icono:

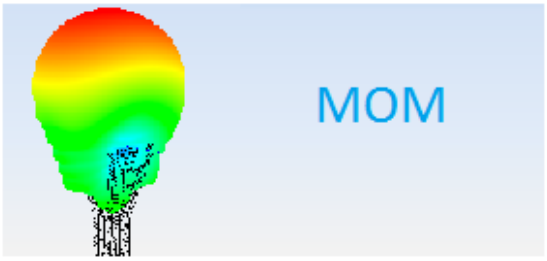

Figura 6.3: Icono del módulo MoM en el panel "New Project"

Paso 3. Ahora se creará un escenario sencillo para la realización de la simulación que constará de una antena de bocina piramidal. Para añadir una antena de este tipo, hay que seleccionar la opción "Antenna → Primitive Antenna → Horns → Pyramidal Horn". Tras hacer esto, aparecerá el siguiente panel, en el que se dejarán los parámetros por defecto salvo en la posición Z, que será 0.2.

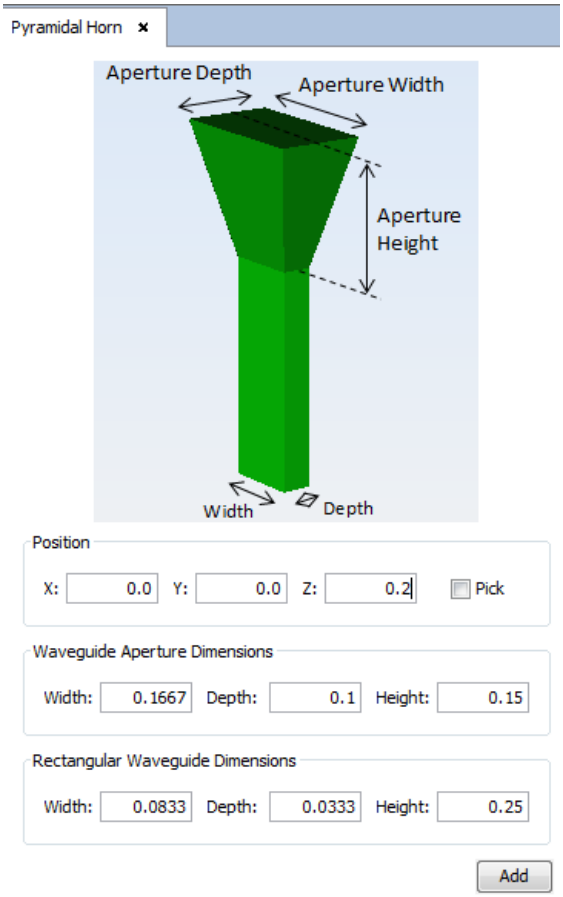

Figura 6.4: Panel de parámetros de la antena de bocina

Tras pulsar el botón “Save”, se creará la nueva antena, la cual se podrá ver en el panel de geometría. La siguiente figura muestra la antena creada:

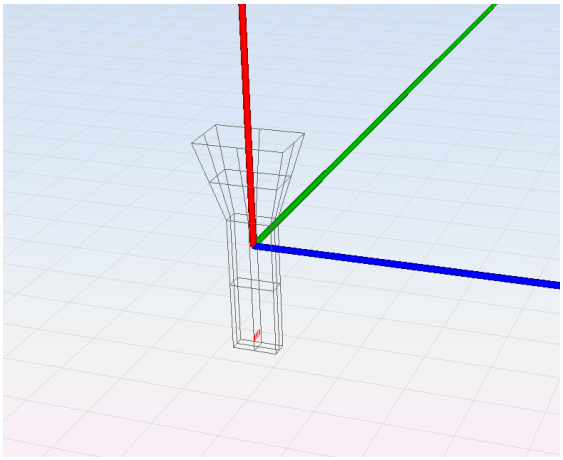

Figura 6.5: Antena en el panel de geometría

Paso 4. Se configurarán ahora los parámetros de la simulación. Para ello, se seleccionará, en la barra de menú, la opción "Simulation → Parameters". Se activa la opción de "barrido de frecuencia", y se asignará un rango de frecuencias que tenga como valor inicial 3.0 GHz y como valor final 4.0 GHz. Se establecerá un número de muestras igual a 2. La configuración de frecuencias quedará, por tanto, como en la siguiente figura:

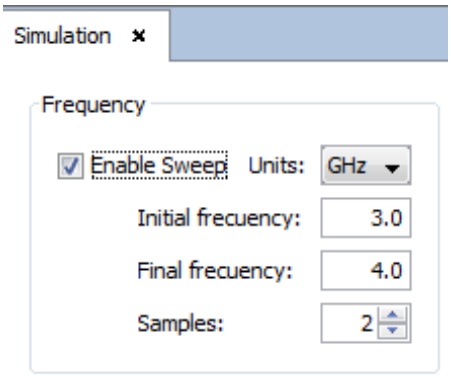

Figura 6.6: Configuración de frecuencias de la simulación

Tras configurar las frecuencias, es necesario pulsar el botón "Save" para confirmar los cambios.

Paso 5. Como último paso de configuración de la simulación, se configurarán las direcciones de observación, que se tendrán en cuenta para el cálculo del campo lejano. Para ello, es necesario seleccionar la opción "Output → Observation Directions". En el

panel “Observation Directions”, se configurará un corte en Phi de un valor inicial de 0.0, un incremento de 1.0 y 91 pasos, tal y como se muestra en la siguiente figura:

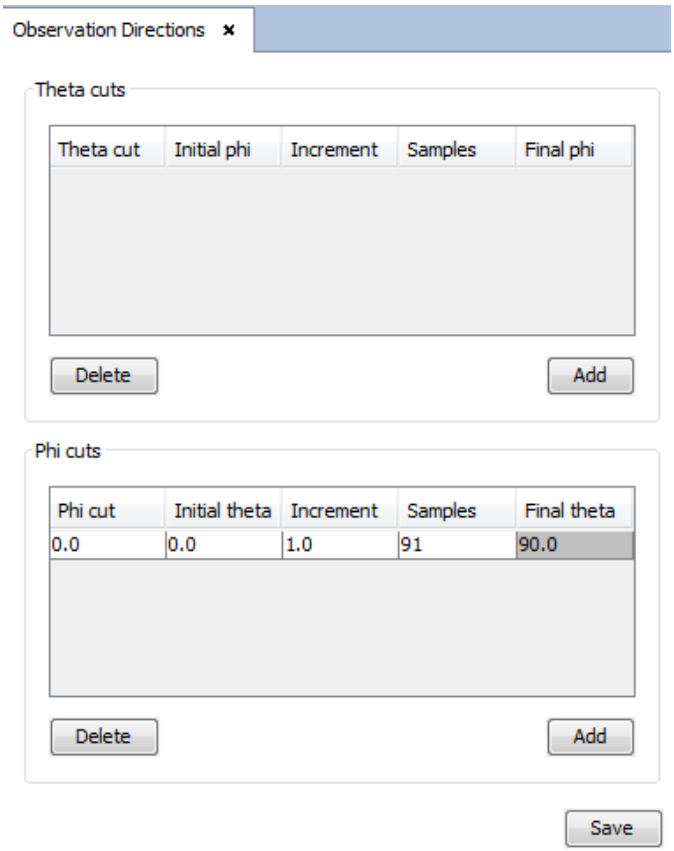

Figura 6.7: Configuración de las direcciones de observación

Al igual que en los pasos anteriores, hace falta presionar el botón “Save” para confirmar los cambios.

Paso 6. Ahora, si se desea ejecutar la simulación en remoto, será necesario configurar, en el cliente, el servidor que será utilizado. Para ello, en la barra de menús, seleccionamos a la opción “Tools → Remote”:

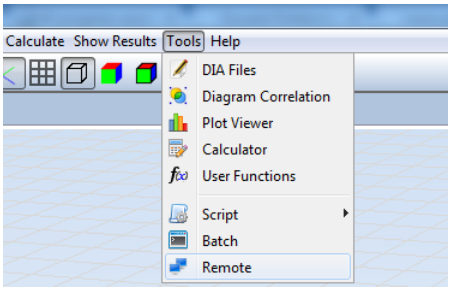

Figura 6.8: Opción de configuración del remoto

En el panel que aparece, se pulsa en el botón “New Server...”. Esto mostrará una ventana emergente en la cual se podrá introducir la dirección IP del equipo remoto y el número de puerto en el que escucha el proceso servidor (por defecto, 3107). Además, de forma opcional, se puede dar un nombre identificativo al servidor. En esta prueba, el servidor tiene la IP 192.168.69.244 por lo que la configuración quedaría como sigue:

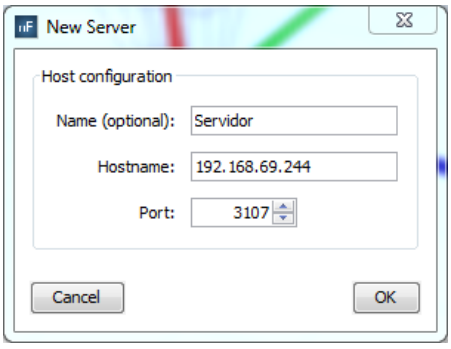

Figura 6.9: Configuración del servidor remoto

Se presiona el botón “OK” para añadir el nuevo servidor a la lista. Tras hacer esto, la configuración quedará como sigue:

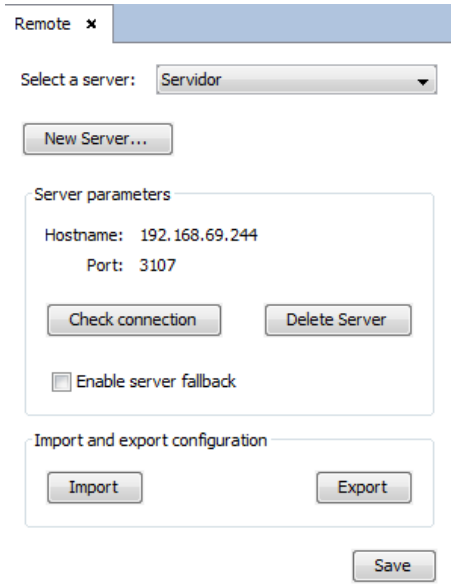

Figura 6.10: Configuración final del remoto

Es posible comprobar que los datos son correctos y que el servidor que se ha configurado está en marcha haciendo clic sobre el botón "Check connection". Tras hacer esto, si se han seguido los pasos anteriores, aparecerá una ventana como la siguiente:

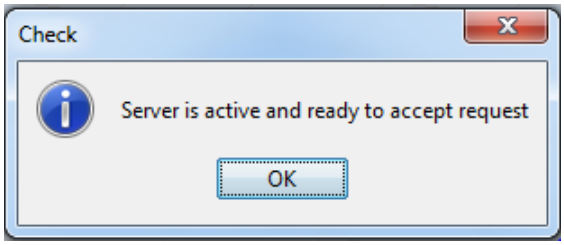

Figura 6.11: Ventana de diálogo informándonos de que el servidor está disponible

Por último, es necesario pulsar el botón "Save" para guardar los cambios, asegurándose previamente de que el servidor que acabamos de añadir se encuentra seleccionado en la combobox de la parte superior del panel (así, las simulaciones que se ejecutarán se harán en remoto usando este servidor).

Paso 7. Una vez se ha configurado el servidor, es posible ejecutar el mallado en remoto. Para ello, se selecciona la opción de menú "Meshing → Create Mesh". Se dejarán las opciones por defecto, como se muestra en la figura:

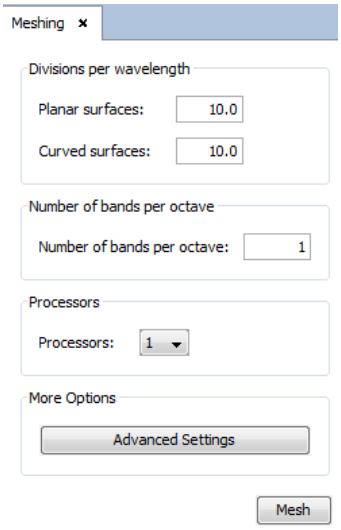

Figura 6.12: Configuración de los parámetros de mallado

Se presiona el botón “Mesh”, lo que iniciará el proceso de mallado remoto. El panel de la derecha de la ventana de la aplicación se mostrará el “Process Log”, que al ser una simulación remota irá detallando los distintos pasos de la comunicación entre cliente y servidor, así como el log de proceso que va generando el servidor y éste envía al cliente. En la captura siguiente se muestra este log de proceso:

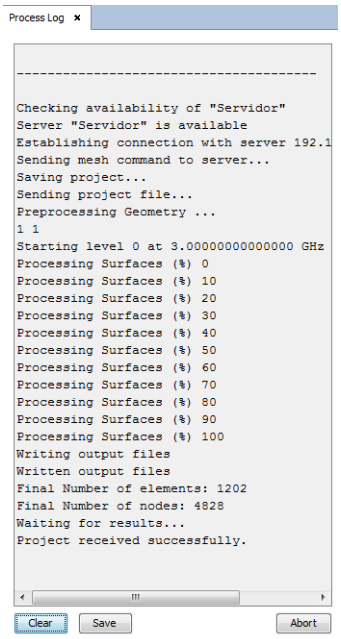

Figura 6.13: Log de proceso mostrado durante el proceso de mallado remoto

Paso 8. Cuando haya terminado el proceso de mallado y el servidor haya enviado los resultados de vuelta al cliente, se puede iniciar el proceso de cálculo de resultados. Para ello, se selecciona la opción "Calculate → Execute" y se presiona el botón "Execute". Esto iniciará la conexión con el servidor remoto para la realización de la ejecución en remoto. El log de proceso mostrará la evolución de los cálculos realizados en el servidor:

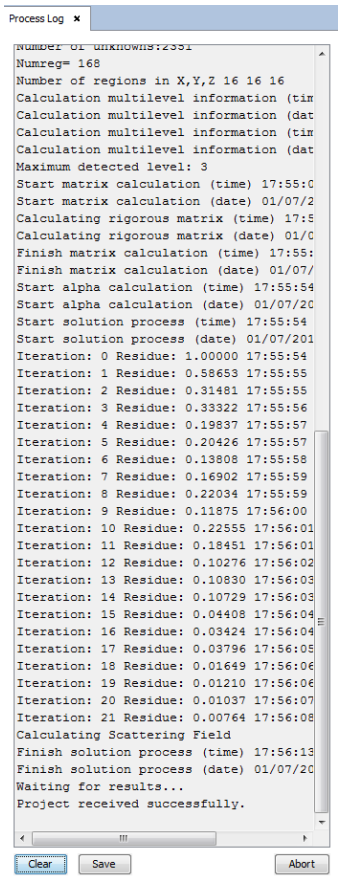

Figura 6.14: Log de proceso mostrado durante el proceso de cálculo de resultados remoto

Una vez ha terminado el cálculo de resultados (proceso que puede tardar varios minutos) se dispondrá de los resultados, disponibles para su visualización.

Paso 10. Se visualizarán algunos de los resultados calculados. Para ello, primero se seleccionará la opción "Show Results → Far Field → View Cuts". Hacer esto mostrará la siguiente gráfica mostrando los valores del campo lejano en cada una de las direcciones de observación especificadas en el Paso 5:

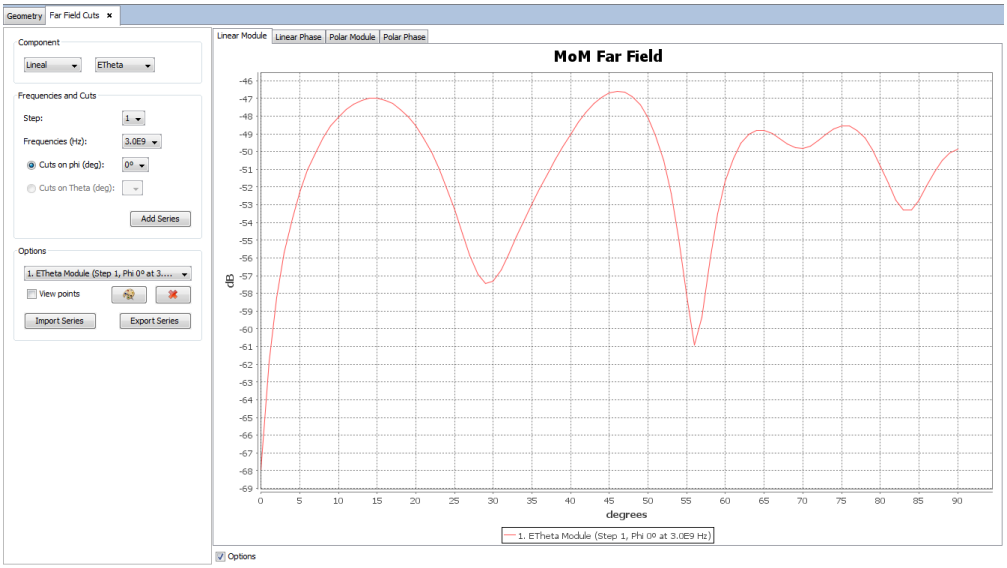

Figura 6.15: Resultados del cálculo del campo lejano

A continuación, se visualizará el patrón de radiación. Para ello, se seleccionará la opción "Show Results → Radiation Pattern → View Cuts". Esto mostrará una gráfica como la siguiente:

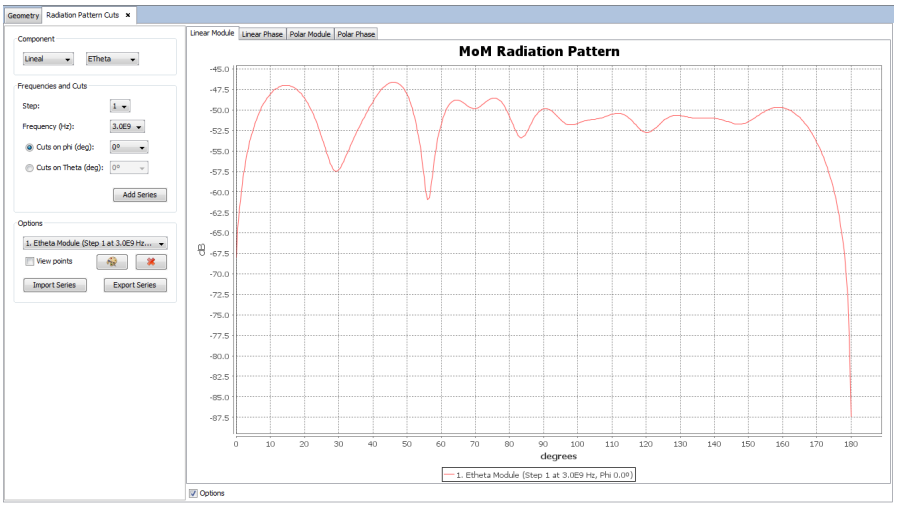

Figura 6.16: Resultados del cálculo del patrón de radiación

Ahora, se visualizarán las densidades de corrientes inducidas sobre la antena de bocina creada. Para ello, se seleccionará la opción "Show Results → View Currents". Esto mostrará el siguiente diagrama 3D:

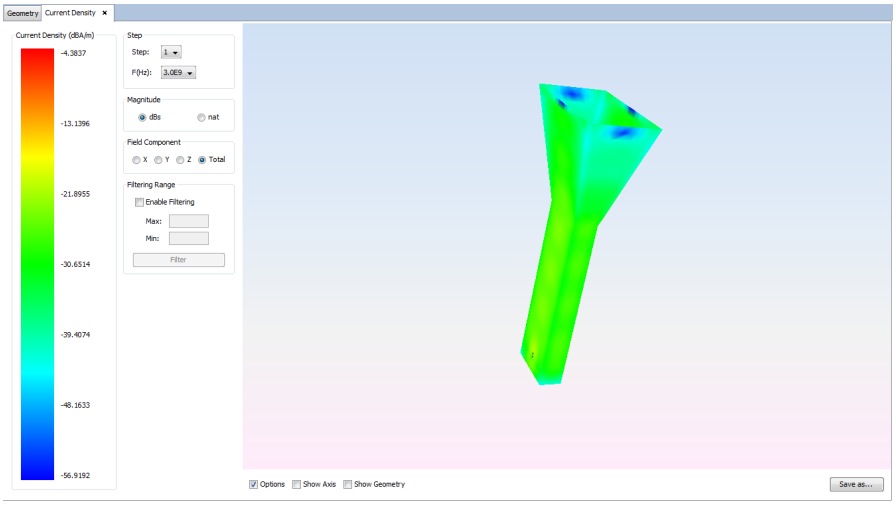

Figura 6.17: Resultados del cálculo de las corrientes inducidas

Por último, se puede ver un diagrama 3D que representa el patrón de radiación generado por la antena. Para esto, es necesario seleccionar la opción "Show Results → Radiation Pattern → View 3D Pattern". Hacer esto mostrará el siguiente panel con el diagrama 3D:

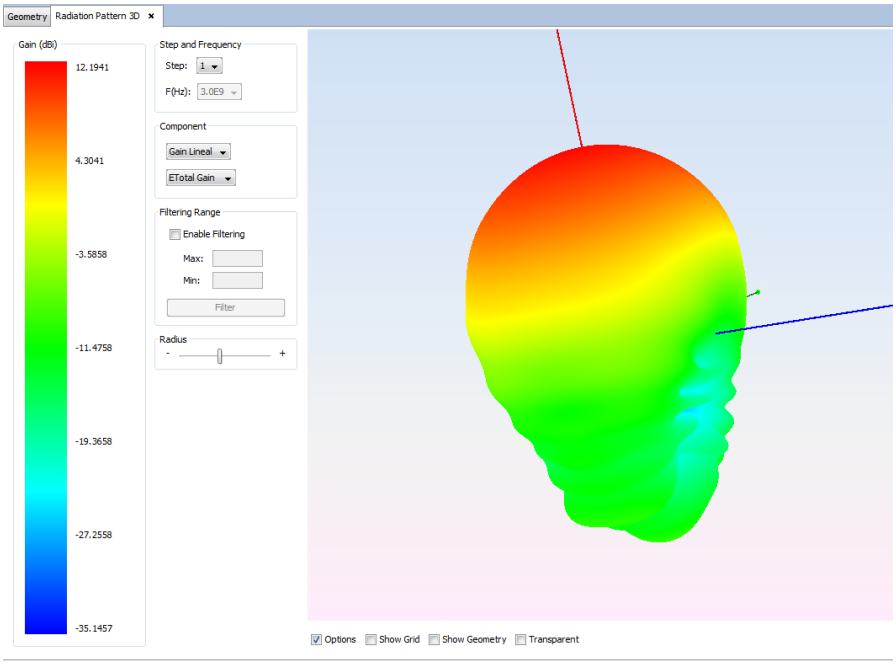

Figura 6.18: Diagrama 3D mostrando el patrón de radiación

Capítulo 7

Manual de mantenimiento

En este capítulo se hablará de las acciones a tomar cuando, en un futuro, se desarrolle un nuevo módulo y se desee ofrecerle soporte para la realización de simulaciones remotas. Será en este capítulo donde se mostrará la facilidad que ofrece el diseño de la aplicación desarrollada para que pueda ser adaptada a otros módulos, tanto de RCS como de antenas. A continuación se muestran los pasos necesarios para lograr este objetivo.

7.1 Adaptaciones en el cliente

En la parte cliente es necesario añadir, en las clases `MeshingParameters` y `CalculateParameters` del módulo, un punto de entrada a la funcionalidad de simulaciones remotas.

Para ello, en `MeshingParameters`, hay que buscar la línea donde se crea y ejecuta el objeto `ExecuteMeshing`. Esta línea deberá asemejarse a la siguiente:

```
mainFrame.processingCommand.execute(new ExecuteMeshing(mainFrame));
```

En este estado, el código solo sería capaz de realizar simulaciones de forma local, independientemente de si hay algún servidor remoto seleccinado. Para permitir las simulaciones remotas, basta con adaptar el código anterior al siguiente:

```
if (mainFrame.getRemoteConfig().isLocalSelected()) {
    mainFrame.processingCommand.execute(new ExecuteMeshing(mainFrame));
} else {
    mainFrame.processingCommand.execute(new ExecuteRemoteMeshing(mainFrame));
}
```

De esta forma, se comprobará si está seleccionada la opción de realizar las simulaciones en local y, en ese caso, se ejecutará el objeto `ExecuteMeshing` (encargado de realizar el mallado en la máquina local). En otro caso, se ejecutará el objeto `ExecuteRemoteMeshing`, el cual se encarga de solicitar el mallado al servidor remoto.

De la misma forma, es necesario realizar esta misma modificación con respecto al cálculo de resultados. Para ello, se busca la siguiente línea en la clase `CalculateParameters` del módulo a adaptar:

```
mainFrame.processingCommand.execute(new ExecuteKernel(mainFrame));
```

Y se sustituye por el siguiente condicional, análogo al mostrado para el proceso de mallado:

```
if (mainFrame.getRemoteConfig().isLocalSelected()) {
    mainFrame.processingCommand.execute(new ExecuteKernel(mainFrame));
} else {
    mainFrame.processingCommand.execute(new ExecuteRemoteCalculate(mainFrame));
}
```

De esta forma, se habilita el módulo para que sea capaz de solicitar operaciones de simulación remotas al servidor. No obstante, el servidor debe ser adaptado para ser capaz de responder a esas peticiones. Esta adaptación se explicará en la siguiente sección.

7.2 Adaptaciones en el servidor

Para hacer funcional la parte servidor de la aplicación, basta con modificar la clase `Project` y, más concretamente, los métodos `mesh()` y `calculate()` de forma que devuelvan un `ProcessingThread` (`ExecuteKernel` o `ExecuteMeshing`) correspondiente al módulo nuevo. Para ello, en la estructura `switch` del método `mesh()`, es necesario añadir un nuevo `case` con la siguiente forma:

```
case TIPO_PROYECTO:
    return new Modules.NewModule.Frames.Meshing.ExecuteMeshing(mainFrame);
```

Donde `TIPO_PROYECTO` es el valor del tipo enumerado (`ProjectType`) que representa el tipo de proyecto, y `NewModule` es el subpaquete que contiene las clases correspondientes al nuevo módulo.

Análogamente, en el `switch` del método `execute()` se debe añadir el siguiente `case`:

```
case TIPO_PROYECTO:
    return new Modules.NewModule.Frames.Calculate.ExecuteKernel(mainFrame);
```

De esta forma, tanto el método `mesh()` como el método `execute()` podrán detectar los proyectos del nuevo módulo y, por tanto, el servidor será capaz de obtener la tarea a realizar para poder realizar las simulaciones solicitadas por el cliente.

Capítulo 8

Conclusiones y posibles líneas de trabajo futuro

8.1 Conclusiones

En este proyecto se ha visto que adaptar una aplicación para que soporte un *workflow* cliente-servidor y conseguir un diseño orientado a objetos que sea sencillo y a la vez limpio no es una tarea baladí. Esto se debe a que el proceso de desarrollo de un sistema software se encuentra lleno de *trade-offs*; es decir, en muchos casos se deben sacrificar algunos beneficios para obtener otros que se consideren más ventajosos (como por ejemplo, complicar ligeramente el diseño para hacer que sea más flexible). Creo que un buen diseñador de sistemas debe ser capaz de hacer las elecciones correctas para escoger aquellos beneficios más adecuados según el sistema que se esté diseñando.

Por otro lado, se debe tener en cuenta que el diseño de una funcionalidad de un software ya existente está condicionado al diseño de la aplicación base, ya que deben "encajar" ambos diseños. En este sentido, es necesario conocer ampliamente como se encuentra estructurada la aplicación antes de proceder a pensar en el diseño y, aun así, es posible pasar muchos aspectos por alto que puedan generar defectos en la aplicación. Se cree que una detección a tiempo de estos defectos es crucial, antes de que esos defectos pasen a producción.

Además, este proyecto también ha ayudado a pensar en una forma de enfocar el desarrollo de una aplicación distribuida cliente-servidor usando el lenguaje de programación Java y ver las dificultades inherentes a este tipo de desarrollos, como la gestión de la concurrencia y de los errores propios de las conexiones de red. Cabe destacar que no disponía de mucha experiencia personal previa en este tipo de desarrollos.

Por último, trabajar sobre un sistema tan complejo como lo es la herramienta *newFASANT* ayuda a abrir la mente sobre como funciona el desarrollo de software en aplicaciones empresariales.

8.2 Posible trabajo futuro

A pesar de todo, se opina que la funcionalidad desarrollada podría hacer uso de algunas cuantas mejoras que mejoren tanto la experiencia del usuario como la funcionalidad del propio sistema. A continuación se mencionan algunas de ellas:

- **Mejora del log del servidor**: Tal y como se encuentra ahora la aplicación, la salida del servidor por la consola es la salida de la propia aplicación y, en lo que respecta al servidor, solo se muestran mensajes cuando se inicia el servidor y se acepta alguna petición de un cliente. Una mejora sería dar la posibilidad de añadir más verbosidad a la salida del servidor, de forma que pueda mostrar datos como: ubicación de red del cliente conectado, tipo de operación y tipo de proyecto solicitado, progreso en la realización de la operación de simulación y, en su caso, errores ocurridos.
- **Posibilidad de trabajo paralelo en el servidor**: Actualmente, un servidor solo es capaz de realizar una simulación en un momento determinado, lo que obliga a disponer de varios servidores si un grupo de trabajo necesita ejecutar varias simulaciones a la vez. Esto no sería necesario si un servidor pudiera realizar más de una simulación en paralelo, aunque en este caso sí podría ser necesario algún tipo de balanceo de carga de forma que las simulaciones no se lancen siempre contra el mismo servidor si hay varios disponibles.
- **Encolado y monitorización de procesos en el servidor**: En el caso de que la mejora del punto anterior no fuera posible, sería interesante que un servidor sea capaz de disponer de una cola de solicitudes pendientes que pudiera ir atendiendo en algún orden. De esta forma, aunque solo se disponga de un servidor y éste no fuera capaz de ejecutar varias simulaciones en paralelo, los clientes podrían enviar los trabajos a realizar al servidor y recibir los resultados cuando estuvieran disponibles (de forma similar al funcionamiento de una impresora). Por otro lado, sería interesante disponer de una interfaz en el servidor (la cual podría incluso ser web) que permita visualizar los trabajos encolados y poder gestionarlos de forma intuitiva (p.ej: cancelarlos o cambiar su prioridad).

Apéndice A

Pliego de condiciones

En este capítulo se establecerá el pliego de condiciones del proyecto desarrollado y de la herramienta *newFASANT*.

A.1 Condiciones generales

La aplicación *newFASANT*, que es sobre la cual se ha trabajado en este proyecto de fin de grado, es una herramienta de software propietario que está disponible en dos modalidades: versión de evaluación y versión completa.

La **versión de evaluación** es gratuita y se permite su uso durante 45 días desde su adquisición, pero debe ser solicitada a través de un formulario disponible en el sitio web oficial ([4]), en el cual el solicitante deberá rellenar sus datos profesionales y personales. Esta versión dispondrá de todos los módulos de la herramienta aunque solo permitirá crear proyectos con geometrías de hasta un número limitado de superficies. Entre las versiones de prueba disponibles se encuentran builds para sistemas Windows tanto de 32 bits como de 64 bits, así como para sistemas Linux de 32 bits y 64 bits [4].

Por otro lado, la **versión completa** es de pago y debe ser solicitada poniéndose en contacto directamente con *NewFasant S.L.*. En este caso, la herramienta será proporcionada al usuario hecha a medida, con solo aquellos módulos de la aplicación que él desee, y los cuales podrá utilizar sin restricciones de funcionalidad. La funcionalidad de simulación remota estará disponible para aquellos módulos que se soliciten.

A.2 Especificaciones técnicas

La aplicación *newFASANT* requiere de unos requisitos mínimos para su ejecución. Estos requisitos mínimos son los siguientes [4]:

- **Sistema operativo**: Linux o Microsoft Windows XP Service Pack 2 (SP2) o superior.
- **Capacidad de memoria principal**: 1 GB de memoria RAM, aunque se recomienda tener 2 GB o más.
- **Arquitectura del procesador**: Intel Pentium Core Duo o superior / AMD Athlon FX o superior.
- **Espacio en disco**: 1 GB o más.
- **Tarjeta gráfica**: NVIDIA GeForce 8 Series o superior / ATI Radeon HD 2000 Series (o superior).
- **Resolución de pantalla mínima**: 1024x768.

Nótese que estos requisitos son los mínimos recomendados para poder ejecutar la aplicación. Si se desea llevar a cabo simulaciones muy complejas, es recomendable que la máquina disponga de unas características más decentes. No obstante, si se desea utilizar la funcionalidad desarrollada, solo será necesario disponer de los recursos de computación de altas prestaciones en el servidor, mientras que el cliente puede permitirse tener especificaciones más modestas.

En cuanto a especificaciones software, es necesario disponer de un sistema Windows igual o superior al indicado o un sistema Linux compatible (se debe contactar con la empresa para confirmar si el sistema Linux que se posee es compatible). Dado que la aplicación ha sido desarrollada usando la versión 1.8 de Java (más conocida como “Java 8”), es necesario disponer del entorno de ejecución de Java 8 (JRE 8) instalado en el sistema.

Apéndice B

Presupuesto

En este capítulo se detallará el presupuesto del proyecto, haciendo una distinción entre diferentes conceptos: coste de hardware y materiales, coste del software utilizado y coste de recursos humanos. Finalmente, se hará un total mostrando los conceptos anteriores, calculando la suma total que representará el coste del proyecto.

B.1 Coste de hardware y materiales

En la Tabla B.1 se detallan el coste del hardware utilizado en el desarrollo del proyecto y su coste. En los casos en los que no se pueda conocer a ciencia cierta el valor del elemento, se dará un valor aproximado:

Tabla B.1: Tabla de presupuesto hardware

Componente	Coste
Procesador Intel Core i5-5287U 2,7GHz	—
Memoria RAM LPDDR3 8 GB (2x4GB) 1866 MHz	—
Tarjeta gráfica Intel Iris Graphics 6100, 1536 MB VRAM	—
Pantalla Retina 13,3”	—
Batería Ión de Litio de 6581 mAh	—
Disco Apple SSD SM0256G	—
Apple Force Trackpad	—
Ordenador Apple Macbook Pro 13,3” (Early 2015)	1.475,79€

Continúa en la siguiente página...

Componente	Coste
Procesador Intel Core 2 Quad CPU Q9400 2,66GHz	—
Memoria RAM 8 GB	—
Tarjeta gráfica NVIDIA GeForce 9400 GT	—
Monitor LG 1920x1080	—
Teclado USB Logitech	—
Disco duro de 500GB	—
Ordenador de sobremesa de la oficina	1.250,00€
Ratón óptico Logitech	8,56€
Coste total hardware	2.734,35€

Por otro lado, como coste de materiales, hay que tener en cuenta el coste de impresión de 3 copias del presente documento, lo cual se estima por unos 90€.

B.2 Coste del software

El coste del software utilizado no es muy elevado debido al uso, en gran medida, de herramientas gratuitas o de código abierto. En la Tabla B.2 se detalla el presupuesto software.

Tabla B.2: Tabla de presupuesto software

Componente software	Coste
Microsoft Windows 7 Professional	134,99€
Mac OS X 10.10.3 (Yosemite)	—[1]
IDE Netbeans 8.0.2	0.00€
Kit de desarrollo de Java 8 (JDK 8)	0.00€

Continúa en la siguiente página...

[1]Incluido en el precio del portátil

Componente software	Coste
Distribución LaTeX “MacTeX”	0,00€
Editor LaTeX “TeXMaker”	0,00€
Coste total software	134,99€

B.3 Coste de recursos humanos

En esta sección se mostrará el coste del trabajo humano, descompuesto en cada una de las tareas realizadas durante el ciclo de desarrollo del software. En la Tabla B.3 se mostrará el cálculo del coste para cada una de las tareas usando un coste por hora estándar y el número de horas que se ha dedicado a cada una. Los costes y tiempos de la tabla incluyen la redacción del presente documento.

Tabla B.3: Tabla de presupuesto de recursos humanos

Tarea	Coste por hora	Número horas	Coste total
Estudio previo de la aplicación	10,00€	30	300,00€
Especificación y análisis de requisitos	15,00€	60	900,00€
Diseño del sistema	30,00€	120	3.600,00€
Implementación y pruebas	15,00€	100	1.500,00€
Desarrollo del resto de documentación	15,00€	90	1.350,00€
Coste total de recursos humanos			7.650,00€

B.4 Presupuesto total

Para finalizar, en la Tabla B.4 se resumen los conceptos dados en las secciones anteriores, calculándose el presupuesto total como la suma de todos los conceptos.

Tabla B.4: Tabla de presupuesto total

Concepto	Coste total
Presupuesto de hardware	2.734,35€
Presupuesto de materiales	90,00€
Presupuesto de software	134,99€
Presupuesto de recursos humanos	7.650,00€
Presupuesto del proyecto	10.609,34€

Apéndice C

Manual de usuario

En este apéndice se dará un manual de usuario explicando las diferentes opciones disponibles, tanto de la aplicación cliente como de la aplicación servidor. Este apéndice se dedicará a hablar de las opciones, por lo que si se desea ver un ejemplo práctico de uso de la aplicación distribuida desarrollada en este proyecto de fin de grado, se recomienda consultar el Capítulo 6.

C.1 Manual del cliente

Si se ejecuta la aplicación *newFASANT* de la manera convencional (haciendo doble clic en el acceso directo que habrá generado el asistente de instalación de la herramienta) y se selecciona la opción "Tools → Remote" desde la barra de menús, aparecerá el panel de la Figura C.1.

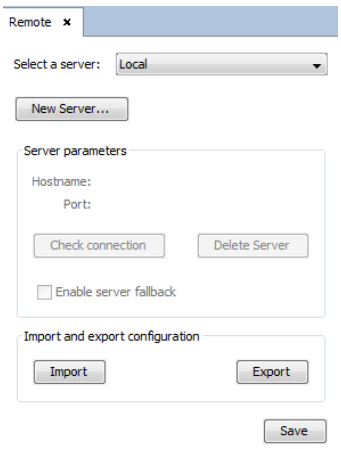

Figura C.1: Panel de configuración del remoto

Desde este panel, se pueden realizar diversas opciones:

- *Seleccionar un servidor*. En la lista desplegable de la parte superior del panel se podrá seleccionar alguno de los servidores dados de alta con el fin de realizar las simulaciones de forma remota en dicho servidor. También se da la posibilidad de seleccionar la opción de realizar las simulaciones de forma local.
- *Añadir un servidor*. Si se presiona el botón "Add Server", se mostrará la ventana de diálogo de la Figura C.2. En esta ventana se podrá dar un nombre para la conexión, así como un nombre de host y un número de puerto obligatorios. Al hacer clic en "OK" se añadirá el nuevo servidor y se podrá seleccionar en la lista.

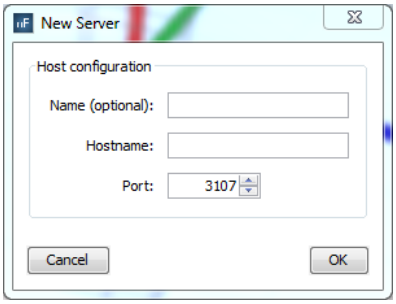

Figura C.2: Ventana de diálogo para añadir servidor

- *Comprobar la disponibilidad del servidor seleccionado*: Haciendo clic en el botón "Check Connection", solo disponible cuando hay un servidor seleccionado, permitirá comprobar la disponibilidad del servidor en la máquina remota. Si el servidor seleccionado está disponible, deberá aparecer la ventana de diálogo de la Figura C.3.

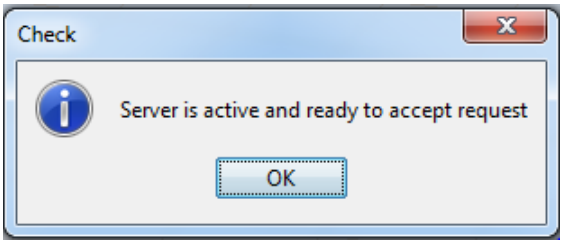

Figura C.3: Ventana de diálogo mostrando disponibilidad del servidor

- *Eliminar el servidor seleccionado*: Haciendo clic en el botón "Delete Server", disponible solamente cuando hay un servidor seleccionado, se borrará ese servidor de la lista de servidores y se seleccionará otro de forma automática.
- *Importar y exportar configuración*: Mediante los botón "Importar", el usuario puede importar un fichero XML conteniendo la configuración del remoto, incluyendo lista de servidores, servidor seleccionado y estado de la opción "server fallback".

Asimismo, el usuario puede exportar la configuración actual hacia un fichero XML ubicado donde el desee pulsando el botón "Exportar". En ambos casos, se abrirá una ventana de diálogo preguntando al usuario por el fichero de origen o de destino.

El usuario tiene también la opción de activar o desactivar la opción de "server fallback". Si está activada, se realiza una simulación y el servidor seleccionado no está disponible, la aplicación intentará realizarla en otros servidores de la lista distintos al seleccionado.

Por último, destacar que al realizar una operación de simulación, es posible abortar la simulación remota desde el panel del log de proceso haciendo clic en el botón "Abort", lo cual cerrará la conexión con el servidor y abortará el proceso de mallado o cálculo en el servidor.

C.2 Manual del servidor

En esta sección se hablará de como ejecutar el servidor y de como crear un acceso directo de forma que el proceso de ejecución sea mucho más cómodo para el usuario.

C.2.1 Ejecutando desde la línea de comandos

Para ejecutar el servidor desde la línea de comandos basta con abrir una consola de Windows, o una terminal si se está en un sistema UNIX, navegar al directorio del ejecutable de la aplicación (mediante la orden `cd`) y ejecutar el siguiente comando:

```
NewFasant6.exe -server
```

De forma opcional, se puede especificar un número de puerto al final del comando (p. ej: `NewFasant6.exe -server 6060`). Si no se indica ningún número de puerto, se usará el puerto 3107.

De esta forma, se iniciará el servidor y podrá atender peticiones de aplicaciones en modo cliente en el puerto correspondiente.

C.2.2 Creando un acceso directo

No obstante, realizar el proceso anterior de abrir una terminal, navegar al directorio de la aplicación y ejecutar el comando puede hacerse muy monótono si se tiene que ejecutar el servidor con mucha frecuencia. Para facilitar el trabajo, se explicará la forma de crear un acceso directo o script que se pueda ejecutar solo realizando doble clic.

Creando un acceso directo: Para crear un acceso directo, basta con hacer clic derecho en el escritorio o directorio donde se desee ubicar, e ir a la opción "Nuevo →

Acceso Directo". Aparecerá la ventana de la Figura C.4 solicitando la ubicación del elemento. En la caja de texto, bastaría con escribir la siguiente línea:

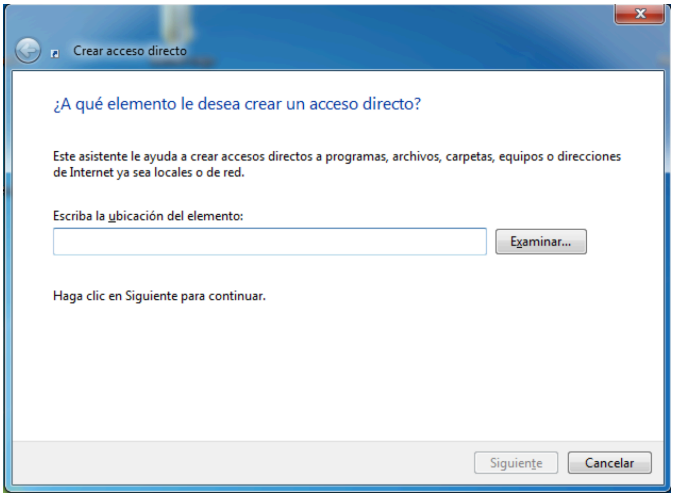

Figura C.4: Ventana de diálogo mostrando disponibilidad del servidor

```
"<ruta al ejecutable>\NewFasant6.exe" -server <puerto>
```

Tras escribir esto, se presiona el botón "Siguiente" y se podrá elegir si dar un nombre al acceso directo. Introducimos el nombre deseado, se presiona "Finalizar" y se creará el acceso directo.

Hecho todo esto, se tendrá un acceso directo para ejecutar la aplicación servidor que se podrá lanzar haciendo doble clic sobre él.

Bibliografía

[1] Romero, G., y Gónzalez, I. (director) (2010), *Diseño e implementación de una plataforma avanzada de ejecución remota vía web: módulos MONURBS y POGCROS* (pp. 14-19), Trabajo Fin de Carrera de Ingeniería en Informática de la Universidad de Alcalá.

[2] Timoteo, L. (2011, 21 de septiembre), *NewFasant, una EBT puntera en el diseño y medición de antenas*, Diario Digital de la Universidad de Alcalá.

[3] Moreno, J., Cátedra, M. F. (director) y Gónzalez, I. (director) (2013), *Desarrollo y optimización de un generador de mallas superficiales y/o volumétricas para aplicaciones de simulación electromagnética*, Tesis Doctoral en la Universidad de Alcalá.

[4] Sitio web oficial de *newFASANT*. `http://www.fasant.com`

[5] Sitio web oficial de Feko. `https://www.feko.info`

[6] Página web de *ANSYS HFSS*. `http://www.ansys.com/es_es/Productos/Flagship+Technology/ANSYS+HFSS`

[7] Sitio web oficial de CST. `https://www.cst.com`

[8] Página web del producto *MMANA-GAL*. `http://hamsoft.ca/pages/mmana-gal.php`

[9] Página web del producto *4nec2*. `http://www.qsl.net/4nec2/`

[10] Coulouris, G., Dollimore, J. y Kindberg, T. (2001), *Sistemas Distribuidos: Conceptos y Diseño*. Addison-Wesley.

[11] Eckel, B. (2002), *Piensa en Java*. Prentice Hall.

[12] Calvert, K. y Donahoo, M. (2008), *TCP/IP sockets in Java*. The Morgan Kaufmann Practical Guide Series.

[13] Ben-Ari, M. (2006), *Principles of concurrent and distributed programming*. Addison-Wesley.

[14] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D. y Lea, L. (2006), *Java concurrency in practice*. Addison-Wesley.

[15] Atomic Access, documentación de Java. https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html

[16] SourceMaking. https://sourcemaking.com